AFRL-IF-RS-TR-2006-352
Final Technical Report
December 2006

QUANTUM COMPUTING AND HIGH PERFORMANCE COMPUTING

General Electric Global Research

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Rome Research Site Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-IF-RS-TR-2006-352 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
EARL M. BEDNAR Work Unit Manager	JAMES A. COLLINS, Deputy Chief Advanced Computing Division Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
DEC 2006	Final	Apr 06 – Oct 06

4. TITLE AND SUBTITLE

QUANTUM COMPUTING AND HIGH PERFORMANCE COMPUTING

5a. CONTRACT NUMBER
FA8750-05-C-0058

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Kareem S. Aggour, Robert M. Mattheyses, Joseph Shultz, Brent H. Allen and Michael Lapinski

5d. PROJECT NUMBER
NBGQ

5e. TASK NUMBER
10

5f. WORK UNIT NUMBER
07

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

General Electric Global Research
1 Research Circle
Niskayuna NY 12309-1027

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFTC
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2006-352

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 06-795

13. SUPPLEMENTARY NOTES

14. ABSTRACT
GE Global Research has enhanced a previously developed general-purpose quantum computer simulator, improving its efficiency and increasing its functionality. Matrix multiplication operations in the simulator were optimized by taking advantage of the particular structure of the matrices, significantly reducing the number of operations and memory overhead. The remaining operations were then distributed over a cluster, allowing feasible compute times for large quantum systems. The simulator was augmented to evaluate a step-by-step comparison of a quantum algorithm's ideal execution to its real-world performance, including errors. To facilitate the study of error propagation in a quantum system, the simulator's graphical user interface was enhanced to visualize the differences at each step in the algorithm's execution. To verify the simulator's accuracy, three ion trap-based experiments were simulated. The simulator output closely matches experimentalist's results, indicating that the simulator can accurately model such devices. Finally, alternative hardware platforms were researched to further improve the simulator performance. An FPGA-based accelerator was designed and simulated, resulting in substantial performance improvements over the original simulator. Together, this research produced a highly efficient quantum computer simulator capable of accurately modeling arbitrary algorithms on any hardware device.

15. SUBJECT TERMS

Quantum Computing, FPGA, Quantum Computer Simulator, Paralelize

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Capt Earl Bednar
U	U	U	UL	71	19b. TELEPHONE NUMBER (*Include area code*)

Table of Contents

Table of Figures

Table of Tables

1.0 PROJECT GOALS

This effort is based on a quantum computer simulator designed and developed by GE Global Research (GEGR) and Lockheed Martin (LM) from 2002 through 2004. The simulator, Quantum eXpress (QX), is capable of accurately simulating any quantum algorithm on any quantum hardware device and is capable of simulating errors from both hardware device imperfections and decoherence.

1.1 Enhance Existing Quantum Computer Simulator

The first objective of this research was to enhance QX, improving its efficiency and increasing its functionality. These enhancements began with researching improvements in the matrix multiplication algorithms used for simulating gate operations. Next, GEGR ported QX to Rome Labs' high performance-computing environment to significantly increase the performance of the system and the number of qubits that can be simulated by exploiting parallelism. Finally, QX was enhanced to simulate, in parallel, the algorithm's execution in both ideal and actual circumstances. Representing the state of the system under both conditions and visualizing the difference at each step can provide important capabilities for studying the effects of errors and error propagation in a quantum system. Increasing the simulator's functionality and improving its performance through these approaches will enable the investigation of error correction schemes, which will allow researchers to quantify the amount of error correction required to develop a large-scale quantum computer with high fidelity.

1.2 Verify Simulator's Accuracy Against Experimental Data

The second goal of this project was to verify the simulator's accuracy by comparing its results to published data from experimental quantum computers. Ion trap quantum computers were chosen for the comparison, for two main reasons. First, they are one of the most promising implementations of quantum computers, with a vigorous research community. Second, the existing Quantum eXpress simulations were based on nuclear magnetic resonance, and testing against ion trap experiments would both be an independent validation and extend the existing library of simulations.

The main steps involved in this validation were:

- Identification of published experiments
- Acquisition of experimental data
- Modeling of the experiments in Quantum eXpress
- Comparison of the simulation and ideal results
- Comparison of the simulation and experimental results

1.3 Simulate Quantum Simulator on an FPGA

The final goal of this effort was to research alternative hardware platforms to improve the performance of the quantum computer simulator. The application of Field Programmable Gate Arrays (FPGAs) to this problem could significantly increase both the speed of quantum simulation runs and the number of qubits

that could be simulated, providing a very promising route to improve the applicability of quantum computer simulation to the general quantum computing research community. GEGR selected a specific FPGA device and associated memory model to be simulated, and then developed an FPGA-based simulation of the Quantum eXpress engine. This FPGA simulation was used to determine efficient data mappings for quantum algorithm simulations, in order to minimize memory contention and gain the greatest possible acceleration in the simulator's performance.

2.0 SIMULATOR OVERVIEW

Quantum Computing (QC) research has gained momentum due to several theoretical analyses that indicate that QC is significantly more efficient at solving certain classes of problems than classical computing [1]. While experimental validation will be required, the primitive nature of today's QC hardware only allows practical testing of trivial examples. Thus, a robust simulator is needed to study complex quantum computing issues. Most QC simulators model ideal operations and cannot predict the actual time required to execute an algorithm, nor can they quantify the effects of errors in the calculation. GE Global Research and Lockheed Martin jointly developed a QC simulator, Quantum eXpress, that models a variety of physical hardware implementations. Quantum eXpress (QX) also allows for the simulation of errors in a quantum computer. Errors typically arise from two sources: 1) hardware device imperfections, and 2) decoherence (the natural tendency of a quantum system to interact with its environment and move from an ordered state to a random state). Both of these sources of error can be simulated.

Most quantum computer simulators are designed to simulate a single algorithm, most commonly Shor's factoring algorithm, on a single type of hardware. QX can be used to implement any quantum algorithm running on any type of hardware, and can report projected algorithm execution times on the quantum device.

QX has a flexible architecture that can be configured entirely through XML files. This enables researchers to explore new algorithms and gate architectures in-silico before they can be physically realized, without having to write any code. QX has been developed entirely in Java 1.4.2 using object-oriented design paradigms. It is platform independent, and has been successfully executed in Windows, UNIX, and Linux environments [2].

2.1 State Representation and the Master Equation

Most quantum computer simulators deal only with pure states and thus cannot accommodate direct error simulation. QX uses the density matrix quantum state representation and time evolution according to a master equation, allowing us to naturally simulate the effects of decoherence. The simulator's ability to accommodate decoherence does come at a price, however. In the density matrix representation, a state of N qubits is represented by a $2^N x 2^N$ square matrix

instead of a 2^N-element vector. Because QX uses the density matrix representation, it cannot handle as many qubits as a pure-state simulator could.

In the absence of decoherence, a state vector (i.e., a general superposition) evolves in time during a single operation according to a Schrödinger equation [3]:

$$i\hbar \frac{d}{dt}|\psi\rangle = H|\psi\rangle, \tag{1}$$

where the matrix *H* is known as a Hamiltonian, which is some linear Hermitian operator, and $\hbar$ is a physical constant known as Planck's constant. An operator (represented as a matrix) is called Hermitian if it is equal to its own transposed complex-conjugate. The vector $|\psi\rangle$, known as a 'ket', is the complex vector associated with state ψ. In the presence of decoherence, and with some approximations, the evolution is described more generally by a "master equation" such as [4]:

$$\hbar \frac{d}{dt}\rho = -i[H, \rho] - [V,[V, \rho]], \tag{2}$$

where square brackets denote commutators (the commutator of two matrix operators A and B is denoted [A,B]) defined as:

$$[A, B] = AB - BA \tag{3}$$

and ρ is a density matrix, a natural way of representing a statistical distribution of states. For a pure state (a completely known superposition), the density matrix has the form [4]:

$$\rho = |\psi\rangle\langle\psi|, \tag{4}$$

where $\langle\psi|$ is the complex conjugate (also referred to as the 'bra') of $|\psi\rangle$. $|\psi\rangle\langle\psi|$ denotes the outer product of the ket and bra. A state remains pure (no decoherence) if V=0 in (2), in which case the master equation is equivalent to the Schrödinger equation from (1). Otherwise, the master equation describes, in a statistical sense, the decohering influence of the environment.

2.2 Evaluating Algorithms in Quantum eXpress

2.2.1 Input

Quantum eXpress requires two primary inputs: (1) a state file and (2) an algorithm file. In the state file a 'base' must be specified, indicating whether the states of the system represent qubits (base 2), qutrits (base 3), or more. While this document will always refer to qubits (2^N), it should be understood that QX can also handle qutrits (3^N) and other higher-order base states, at the user's

discretion. The initial state of the system is represented by a vector of 2^N elements (again, presuming base 2), where N is the number of distinct qubits.

The base and initial states of Quantum eXpress are specified in an eXtensible Mark-up Language (XML) file using the World Wide Web Consortium's (W3C 2001) Mathematical Mark-up Language (MathML) specification. This file contains sets of vectors defining both the initial states and 'states of interest'. These states are effectively identical in construction, except the initial states also have probability values associated with them indicating the probability that the initial system is in that state. States of interest are defined for the purpose of allowing QX to observe certain states. At any time during the execution of an algorithm, the system can be evaluated to determine the probability of it being in each of these observed states. At the end of the execution of an algorithm, the probabilities of each of the states of interest are displayed to give an indication of the final superposition of the system. An excerpt from a state configuration file for a base 2, 3 qubit system can be seen in Figure 1. From this figure, we can see that the vectors for both the initial and interest states are complex with $2^3 = 8$ elements per vector.

```
<quantum-states>
  <class>com.quantum.system.qx.QuantumSystemQX</class>
  <base> 2 </base>
  <qu-number> 3 </qu-number>

  <initial-states>
    <state>       <!-- |000> + |100> -->
      <id> 000 + 100 </id>
      <vector>
       <cn type="complex-cartesian"> 0.70710678118654752 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0.70710678118654752 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
      </vector>
      <probability> 1 </probability>
    </state>
  </initial-states>

  <interest-states>
    <state>       <!-- |000> -->
      <id> 000 </id>
      <vector>
       <cn type="complex-cartesian"> 1 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
```

```
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
      </vector>
    </state>
    <state>       <!-- |001> -->
      <id> 001 </id>
      <vector>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 1 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
       <cn type="complex-cartesian"> 0 <sep/> 0 </cn>
      </vector>
    </state>
    <!-- ... -->
  </interest-states>

</quantum-states>
```

Figure 1: Sample State XML Configuration File

The other required input is a second XML file that describes the quantum algorithm to be executed. The algorithm includes what gate operations to run and on which qubits those operations are performed. This file is maintained separately from the initial state file, so that a single algorithm can be easily executed with various initial states. An example algorithm configuration file for a 3 qubit system can be seen in Figure 2. From this figure, we can see that each gate operates on a specific set of qubits.

```
<quantum-circuit>
  <name>Algorithm Name</name>
  <qu-number>3</qu-number>
  <n>100</n>

  <algorithm>
    <unitary>
      <operator>CNot</operator>
      <a>1</a>
      <qu>1</qu>
      <qu>2</qu>
    </unitary>
    <unitary>
      <operator>CNot</operator>
      <a>2</a>
      <qu>2</qu>
      <qu>3</qu>
    </unitary>
    <unitary>
      <operator>CNot</operator>
      <a>4</a>
      <qu>1</qu>
      <qu>2</qu>
```

```
    </unitary>
  </algorithm>
</quantum-circuit>
```

Figure 2: Sample Algorithm XML Configuration File for 3 Qubit Test Case

Figure 2 repeatedly references a specific unitary operator—the *CNot* gate. The definition of the *CNot*, and any other gate elements that may be referenced in an algorithm, are kept in separate gate configuration files. As an example, the *CNot* gate XML configuration file can be seen in Figure 3. These files are maintained separately from the algorithm so that they can be easily reused. These files include the base of the quantum system, the Hamiltonian to apply, the amount of time the Hamiltonian needs to be applied, and the ideal unitary operator matrix that the Hamiltonian, perfectly applied for the specified time, should produce.

```
<gate>
  <base>2</base>
  <ideal>
    <matrix>
      <matrixrow>
        <cn>1</cn> <cn>0</cn> <cn>0</cn> <cn>0</cn>
      </matrixrow>
      <matrixrow>
        <cn>0</cn> <cn>1</cn> <cn>0</cn> <cn>0</cn>
      </matrixrow>
      <matrixrow>
        <cn>0</cn> <cn>0</cn> <cn>0</cn> <cn>1</cn>
      </matrixrow>
      <matrixrow>
        <cn>0</cn> <cn>0</cn> <cn>1</cn> <cn>0</cn>
      </matrixrow>
    </matrix>
  </ideal>
  <hamiltonian>
    <matrix>
      <matrixrow>
        <cn>0</cn> <cn>0</cn> <cn>0</cn> <cn>0</cn>
      </matrixrow>
      <matrixrow>
        <cn>0</cn> <cn>0</cn> <cn>0</cn> <cn>0</cn>
      </matrixrow>
      <matrixrow>
        <cn>0</cn> <cn>0</cn> <ci> a </ci> <ci> -a </ci>
      </matrixrow>
      <matrixrow>
        <cn>0</cn> <cn>0</cn> <ci> -a </ci> <ci> a </ci>
      </matrixrow>
    </matrix>
  </hamiltonian>
  <time>pi  / (  2  *  a  ) </time>
</gate>
```

Figure 3: Sample Gate XML Configuration File

2.2.2 Algorithm Simulation

Quantum simulators need a succinct method for describing quantum systems and their operations. Since a state is represented as a vector (ket), a statistical ensemble of states is naturally represented as a matrix, referred to as a (probability) density matrix. The density matrix describes the current state of the quantum system. The execution of a quantum algorithm can be viewed as the multiplication of a system's density matrix with other matrices that represent quantum operations.

The initial states and their probabilities determine the initial density matrix of the system using the equation:

$$\rho = \sum_{k=1}^{\#init\ states} p(k) \,|\, k\rangle\langle k \,|\,, \tag{5}$$

where *p(k)* is the probability of state *k*. Equation (5) allows us to define the initial density matrix *ρ* of the system.

A third input into the system is a set of 'gate' XML files that define the structure of these operations. Each gate is a unitary operator, which is defined by a Hamiltonian matrix and a time *Δt* over which it is applied. This is described by the following equation:

$$U(\Delta t) = e^{-iH\Delta t/\hbar}, \tag{6}$$

where *U* is the unitary operator and *H* is the Hamiltonian for the gate.

Simulation Without Decoherence

If we do not simulate decoherence in the master equation in (2), the operators *U(Δt)* are applied to the density matrix *ρ* according to the following equation:

$$\rho(t+\Delta t) = U(\Delta t)\,\rho(t)\,U(\Delta t)^{\dagger}. \tag{7}$$

To minimize the numerical error in the simulation, instead of calculating the density matrix *ρ* once after the full time step *Δt*, QX divides the time step by some value *n*, and repeatedly calculates the next value of *ρ*. Figure 4 shows this function. In Quantum eXpress, *n* is set to 100.

```
for i = 1 to n
```

$$\rho(t+\frac{\Delta t}{n}) = U(\frac{\Delta t}{n})\,\rho(t)\,U(\frac{\Delta t}{n})^{\dagger}$$

Figure 4: Simulation Without Decoherence Function

Here $U(\Delta t)^{\dagger}$ is the Hermitian conjugate of $U(\Delta t)$. The gate XML file contains the matrix H and Δt in MathML format. Each gate may act on a different number of possible qubits, as some apply to single qubits (e.g., Not), some apply to two (e.g., CNot {Conditional Not} and Swap), and so on. The exact Hamiltonian to apply and for how long depends on (a) the type of gate operation and (b) the type of hardware. E.g., a 'Not' gate may have different Hamiltonians depending on the type of hardware modeled.

The exponentiation of the matrix H in (6) is evaluated using the Taylor Series expansion of e^x:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^k}{k!} + ... \tag{8}$$

Combining (6) and (8), the unitary operator U may be written as:

$$U(\Delta t) = e^{-iH\Delta t/\hbar} \approx I - \frac{iH\Delta t}{\hbar} - \frac{1}{2!}(\frac{H\Delta t}{\hbar})^2 + \frac{i}{3!}(\frac{H\Delta t}{\hbar})^3 . \tag{9}$$

Note that the approximation of $e^{-iH\Delta t/\hbar}$ uses the third-order of the Taylor Series expansion. This could be increased to improve the numerical accuracy of the simulator (though it would negatively impact its efficiency). Using the cubic expansion produces numerical errors on the order of 10^{-5}, which for most evaluations is quite sufficient. Equations (5) through (9) illustrate, using the no-decoherence case, how the simulator evaluates quantum algorithms.

Simulating Decoherence

If decoherence is simulated, the master equation in (2) can be represented as:

$$\frac{d}{dt}\rho = -i[H,\rho] + [V\rho, V^H] + [V, \rho V^H], \tag{10}$$

where V is a matrix that represents decoherence being applied to the quantum system. If we assume the V decoherence matrices can be applied to each qubit N in the density matrix independently, then (10) can be represented as:

$$\frac{d}{dt}\rho = -i[H,\rho] + \sum_{j=1}^{N}([V_j\rho, V_j^H] + [V_j, \rho V_j^H]). \tag{11}$$

A second-order discrete-time approximation of (11) can be written as [5]:

$$\rho(t+\Delta t) \approx \left\{1 + \Delta t\ \Omega + \frac{\Delta t^2}{2!}\Omega^2\right\}\rho(t), \tag{12}$$

where

$$\Omega\rho = -i[H,\rho] + \sum_{j=1}^{N}\left([V_j\rho, V_j^H] + [V_j, \rho V_j^H]\right) \tag{13}$$

and

$$\Omega^2\rho = -i[H,\Omega\rho] + \sum_{j=1}^{N}\left([V_j\Omega\rho, V_j^H] + [V_j, \Omega\rho V_j^H]\right). \tag{14}$$

Using the commutator defined in (3) to expand $\Omega\rho$ and $\Omega^2\rho$ we obtain:

$$\Omega\rho = -i(H\rho - \rho H) + \sum_{j=1}^{N}\left(2V_j\rho V_j^H - V_j^H V_j\rho - \rho V_j^H V_j\right) \tag{15}$$

and

$$\Omega^2\rho = -i(H\Omega\rho - \Omega\rho H) + \sum_{j=1}^{N}\left(2V_j\Omega\rho V_j^H - V_j^H V_j\Omega\rho - \Omega\rho V_j^H V_j\right). \tag{16}$$

From the substitution of (15) and (16) into (12) we can determine the complete simulation of a quantum algorithm with decoherence requires 12*N*+4 matrix multiplication at each time step *Δt*, where *N* is the number of qubits in the system. To minimize the numerical error in (12), instead of calculating the density matrix *ρ* once after the full time step *Δt*, QX divides the time step by some value *n*, and repeatedly calculates the next value of *ρ*. Figure 5 shows this function. In Quantum eXpress, *n* is set to 100. Due to these *n* iterations, we find that the evaluation of a quantum algorithm with decoherence requires (12*N*+4)*100 matrix multiplications at each time step.

```
for i = 1 to n
```

$$\rho\left(t + \frac{\Delta t}{n}\right) \approx \left\{1 + \frac{\Delta t}{n}\ \Omega + \frac{1}{2!}\left(\frac{\Delta t}{n}\right)^2\Omega^2\right\}\rho(t)$$

Figure 5: Simulation With Decoherence Function

Decoherence Examples

Two forms of decoherence can be simulated in Quantum eXpress, phase damping and amplitude damping [6, 7].

Phase Damping

When we talk about “measuring” a qubit in the computational basis, we’re talking about a process in which the qubit modifies its macroscopic environment one

way if it is in the state $|0\rangle$ and modifies it a different way if it is in the state $|1\rangle$. If the qubit is in a superposition of these, then unitarity implies that the same interaction causes it to become entangled with the macroscopic environment. The result is either a $|0\rangle$ or a $|1\rangle$, and the original superposition is lost. This is an example of "phase damping."

More generally, phase damping is any interaction between the qubit and its environment in which:

- If the qubit is $|0\rangle$, it affects the environment one way and remains $|0\rangle$
- If the qubit is $|1\rangle$, it affects the environment a different way and remains $|1\rangle$.

An example of such an interaction is:

$$V = \begin{bmatrix} 0 & 0 \\ 0 & b \end{bmatrix} \tag{17}$$

for some constant b. This says that a qubit in state $|0\rangle$ does nothing to its environment, but a qubit in state $|1\rangle$ does something. The matrix *V* is optionally included in the algorithm configuration file. An example phase damping matrix is shown in Figure 6, exactly as it would be included in the configuration file. The absence of such a matrix, or the inclusion of an all-zero decoherence matrix indicates to the simulator that the algorithm should be executed sans decoherence.

```
<decoherence>
  <matrix>
    <matrixrow>
    <cn> 0 </cn> <cn> 0 </cn>
    </matrixrow>
    <matrixrow>
    <cn> 0 </cn> <cn> 0.005 </cn>
    </matrixrow>
  </matrix>
</decoherence>
```

Figure 6: Sample Decoherence Matrix in XML

Amplitude Damping

Suppose that a qubit in state $|1\rangle$ can "decay" into state $|0\rangle$ by emitting a photon. This does two things: first, unlike phase damping, it changes the state of the qubit (unless it was $|0\rangle$ already). Second, like phase damping, it causes $|0\rangle$ and $|1\rangle$ to affect the environment in different ways. Only one of these two states can

emit a photon into the environment. Because of the second effect, this is another example of decoherence. It is called “amplitude damping” because no matter what state we start with, we eventually end up with $|0\rangle$.

An example of an amplitude-damping interaction is:

$$V = \begin{bmatrix} 0 & b \\ 0 & 0 \end{bmatrix} \tag{18}$$

for some constant b. This says that nothing happens to a qubit in state $|0\rangle$, but a qubit in state $|1\rangle$ can change into $|0\rangle$.

Simulating Device Imperfections/Noise

Noise due to device imperfections and other factors is at the gate level, and therefore, there is a different noise element potentially associated with each gate. Noise is simulated in QX by modifying the time with which a gate’s Hamiltonian is applied to the quantum system. This means that the matrices used to simulate an algorithm do not change, and therefore, no new equations are required to simulate noise. QX will use the standard decoherence or decoherence-free equations for simulating the algorithm, and only add Gaussian noise to the gate application times. An example Gaussian noise element (defined by a mean and standard deviation) is shown in Figure 7, exactly as it would be included in the gate configuration file.

```
<noise>
  <mean>0</mean>
  <stddev>0.0005</stddev>
</noise>
```

Figure 7: Sample Noise in XML

2.2.3 Output

At the completion of the evaluation of an algorithm, we wish to understand the final superposition of the quantum system. The states of interest from Figure 1 are measured against the final density matrix to determine the probability that the system is in each state, using the following equation:

$$p(k) = trace \quad (|k\rangle\langle k|\rho), \tag{19}$$

where *p(k)* is the probability that the final superposition is in state *k* described by ket $|k\rangle$.

3.0 SUMMARY OF KEY ACCOMPLISHMENTS

What follows is a bulleted list of the key accomplishments of this project:

- Optimized existing quantum computer simulator's matrix multiplication operations by taking advantage of particular structure of matrices.
- Reduced order of simulator operations per time step from $O(2^{3N})$ to $O(2^{2N+g})$, where N is the number of qubits in the algorithm and g is the number of qubits on which the gate operates.
- Reduced simulator memory overhead requirement from $2^N x 2^N$ to $2^g x 2^g$.
- Achieved a 99.5% performance improvement for a single-processor simulation of 7-qubit Shor's Algorithm with decoherence (from just under two days to just over twelve minutes).
- Further enhanced simulator by distributing simulator matrix calculations across a cluster of at most $2^{2(N-g)}$ nodes.
- Achieved an 87.5% performance improvement over the previously optimized simulation of the 7-qubit Shor's Algorithm with decoherence, using a cluster of 16 nodes (reduced simulation time from just over twelve minutes to a minute and a half).
- Achieved an overall performance improvement of 99.94% from the initial simulator implementation to the optimized and distributed simulation of the 7-qubit Shor's Algorithm with decoherence.
- Enhanced simulator to evaluate a step-by-step comparison of the ideal quantum algorithm execution to the actual (decoherence and/or error-included) simulation, storing the density matrix difference after each gate application.
- Evaluated Frobenius norm to quantify difference between two matrices after each step.
- Augmented simulator graphical user interface to visualize heatmaps of ideal vs. actual density matrices at each step in algorithm.
- Modeled three experimental ion trap quantum computer algorithms: Deutsch-Jozsa, Grover, and semi-classical quantum Fourier Transform.
- Compared the simulator's results in the absence of noise to the ideal ion trap algorithm results, resulting in errors less than 0.002%.
- Compared the simulator's results with gate noise to the experimental ion trap results.
- Achieved overlapping fidelities between the simulator's results and the experimental results with 10% noise in the gate application times.
- Implemented an FPGA-based accelerator for quantum gate application.
- Achieved a 12.4x performance improvement over the single processor implementation for a 3-qubit test case.
- Achieved a 26.9x performance improvement over the single processor implementation for a 5-qubit test case.
- Compared precision loss stemming from using a single precision floating point data type for computations rather than the double precision data type used on the general purpose processors.

4.0 DETAILS OF KEY ACCOMPLISHMENTS

4.1 Performance Enhancement via Matrix Multiplication Optimizations

The objective of these enhancements was to optimize the matrix multiplications in the simulator to improve its performance. A quantum computer simulator that demonstrates interactive performance for small systems and feasible compute times for larger quantum systems would provide a powerful test-bed for exercising and quantifying the performance of quantum algorithms.

4.1.1 Test Cases

Three different algorithms (utilizing 3, 5, and 7 qubits, respectively) were used to evaluate the original performance of the simulator. The 3-qubit algorithm, shown in Figure 8, is comprised of three Conditional Not (CNot) gates.

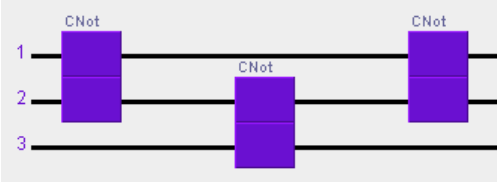

Figure 8: 3 Qubit Test Case

The 5-qubit algorithm, shown in Figure 9, is comprised of six gates; three of which are 2-qubit CNot gates, and the remaining three of which are 3-qubit Conditional-Conditional Not gates (CCNot).

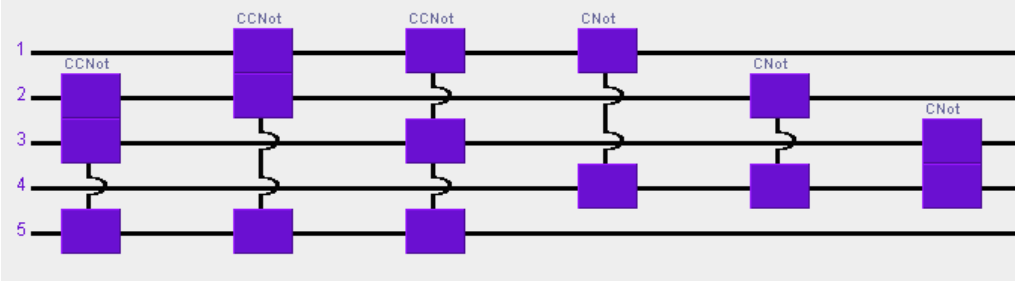

Figure 9: 5 Qubit Test Case

The 7-qubit algorithm used for evaluating Quantum eXpress is Shor's Algorithm for the prime factorization of numbers [1]. Operating on 7 qubits, Shor's Algorithm factors the number 15 into the prime numbers 3 and 5. This algorithm is shown in Figure 10.

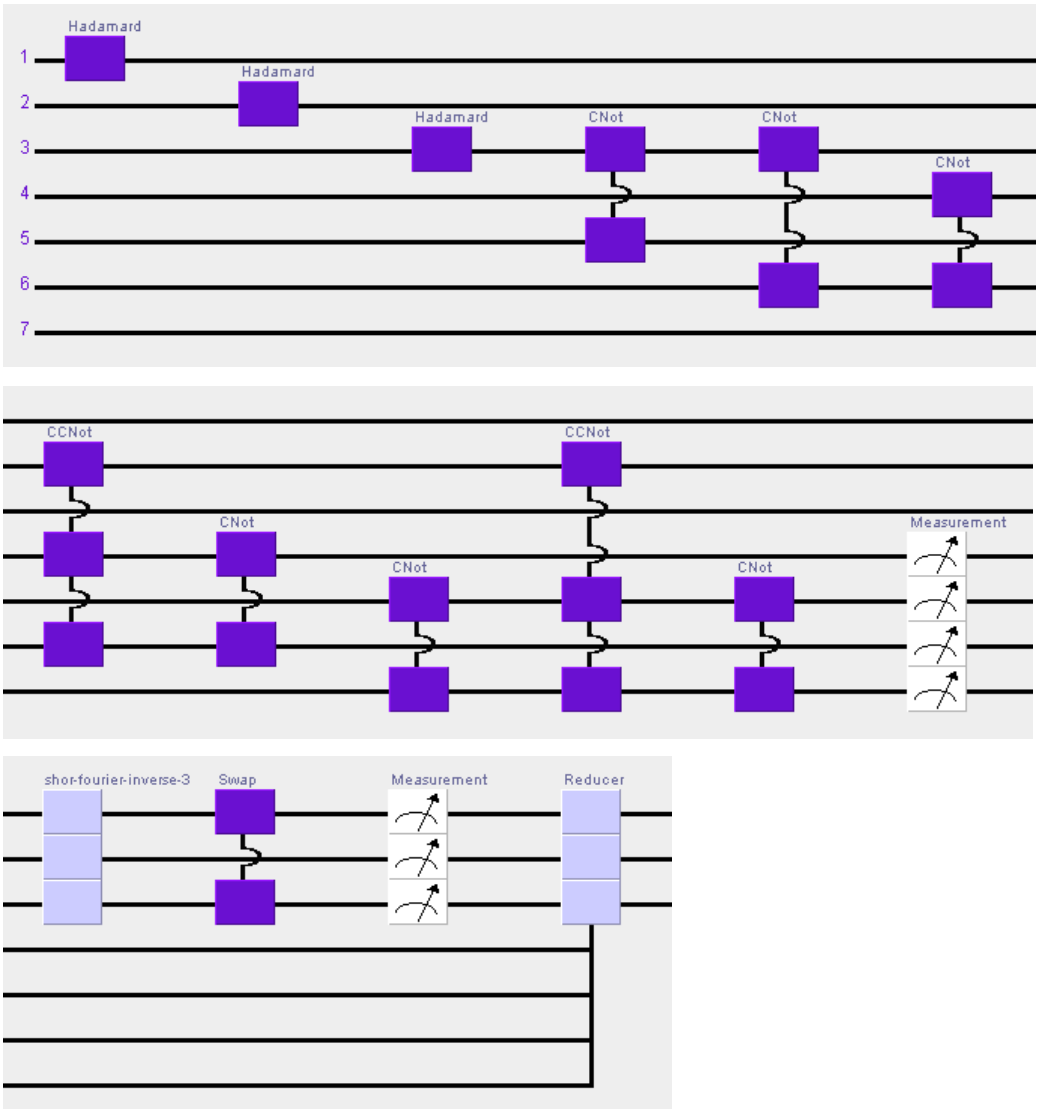

Figure 10: 7 Qubit Test Case (Shor's Algorithm)

As can be seen in Figure 10, a smaller algorithm is incorporated into the primary algorithm ("shor-fourier-inverse-3"). This is a 3-qubit inverse Fourier Transform, which is shown in Figure 11. In total, the 7-qubit algorithm is comprised of 21 gates of various type.

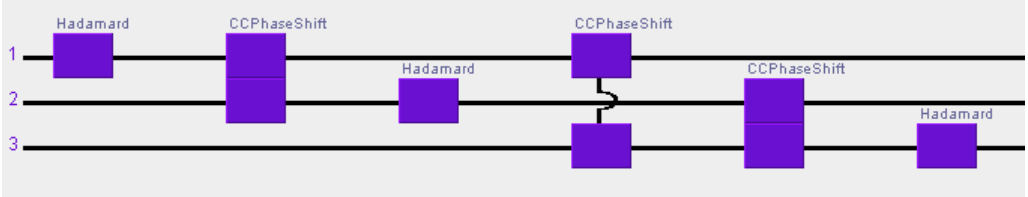

Figure 11: 3 Qubit Inverse Fourier Transform for 7 Qubit Test Case

The results of evaluating these three algorithms with the original implementation of QX can be seen in Table 1. Clearly, the no decoherence implementation exhibits run times that are acceptable for interactive use. The original decoherence simulation times are extremely inefficient, however. Adding a few more qubits would quickly increase run times to weeks or months.

Table 1: Simulation Times Before Optimization

	No Decoherence	Original Decoherence
3 Qubit	2.1s	5.2s
5 Qubit	4.6s	720.6s (12m 0.6s)
7 Qubit	88s (1m 28s)	155191.4s (43h 6m 31s)

4.1.2 Multiplication Optimizations

The core computation performed by the simulator is matrix multiplication. It is costly for two reasons. First, the matrices are large. The density matrix representation of the quantum state requires matrices with the number of rows and columns exponential in the number of qubits being simulated. Thus, to simulate a system with N qubits requires a density matrix with 2^N rows and 2^N columns. For example, for N=8, a 256x256 matrix is required, resulting in 65,536 elements. For N=16, a 65,536x65,536 matrix is required, resulting in over 4.2 billion elements in the matrix. Second, general matrix multiplication is relatively expensive. In order to simulate algorithms with large numbers of qubits, alternative matrix multiplication algorithms are required.

The Strassen Algorithm replaces expensive matrix multiplication operations with less-expensive addition and subtraction operations [8]. Therefore, the Strassen Algorithm with Winograd's Variant was implemented in QX to improve the speed of matrix multiplication over the traditional row-column multiplication algorithm. However, due to the additional memory overhead of Strassen's Algorithm, significant performance improvements were not achieved. In order to realize significant improvements it was necessary to exploit the unique structure of the matrices used in the Quantum eXpress simulations.

In this section we develop an algorithm to perform the single time step update represented by (11). For clarity we will focus on the core, single time step computation that serves as the basis for simulating the application of a gate. Improvements in this computation will lead to proportional improvements in the overall system performance. Although we concentrate on run time, our solution will also provide an improvement in storage requirements.

Our first block of pseudo-code, found in Figure 12, is for the case where there is no decoherence. This is represented by (11), (14) and (15), where the decoherence terms are all zero. A straightforward implementation based on the inner product method for matrix multiplication is given in the first code fragment.

```
ApplyGate1Step(Rho, H, dT)
Inputs:
Rho   a square matrix of side QB representing the state of the quantum
      system being simulated.  The result is returned as an update to
      Rho
H     the same size as Rho, is the Hamiltonian operator for the gate
      being applied
dT    the time step size
LocalVariables:
f1 = dT and f2 = (dT^2)/2
QB    the number of rows (columns) in the matrix Rho
Ts1, Ts2 an array the same size as Rho, which holds the first (second)
      order term of the Taylor series.  They are allocated once and
      reused on subsequent calls
// Compute the first order term (H*Rho - Rho*H)
      for (j = 0; j++; j < QB)
            for (k = 0; k++; k < QB)
                  Ts1[j,k] = 0
                  for (l = 0; l++; l < QB)
                        Ts1[j,k] += H[j,l]*Rho[l,k]
                        Ts1[j,k] -= Rho[j,l]*H[l,k]
// Compute the second order term (H*T1 - T1*H)
      for (j = 0; j++; j < QB)
            for (k = 0; k++; k < QB)
                  Ts2[j,k] = 0
                  for (l = 0; l++; l < QB)
                        Ts2[j,k] += H[j,l]*Ts1[l,k]
                        Ts2[j,k] -= T1[j,l]*H[l,k]
// Update Rho according to Rho + f1*T1 + f2*T2
      for (j = 0; j++; j < QB)
            for (k = 0; k++; k < QB)
                  Rho[j,k] += f1*Ts1[j,k] + f2*Ts2[j,k]
```

Figure 12: Non-Optimized, No Decoherence Matrix Simulation

This code requires $O(2^{3N})$ arithmetic operations per time step and two full temporary matrices the size of the density matrix ρ (Rho). Without more information the above code would be acceptable.

We do know, however, that although ρ is, in general, full, H will be sparse. In fact, H will have a very special structure. To better understand the structure we need to consider the relationship between an index (row or column) and the basis states of the individual qubits. The basis vectors (indices) for a system are formed by interpreting the catenation of the basis vectors for the individual qubits as a binary number where bit position i corresponds to the value of the basis vector for qubit i. For example, 100_2 refers to the state where $\text{qubit}_0 = 0$, $\text{qubit}_1 = 0$, and $\text{qubit}_2 = 1$.

To see this, consider the construction of H, the Hamiltonian for the system, in terms of G, the Hamiltonian of the gate. This relationship is best understood by looking at a specific case. Consider a two-qubit gate. The most general case of

a Hamiltonian, *G*, defining a gate's effect on two qubits, is a full 4x4 matrix shown in (20).

$$G = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}. \tag{20}$$

Now consider the structure of the system Hamiltonian *H(G)* resulting from the application of the gate *G*. Since the system has more qubits than the gate operates on we must choose which qubits to transform and which one to leave untouched. For the purpose of this example we will apply the gate to $qubit_0$ and $qubit_1$ leaving $qubit_2$ untransformed. This produces the system Hamiltonian:

$$H(G) = \begin{pmatrix} a & b & c & d & & & & \\ e & f & g & h & & 0 & & \\ i & j & k & l & & & & \\ m & n & o & p & & & & \\ & & & & a & b & c & d \\ & 0 & & & e & f & g & h \\ & & & & i & j & k & l \\ & & & & m & n & o & p \end{pmatrix}. \tag{21}$$

The upper left block of *H(G)* corresponds to the transformation on states corresponding to $qubit_2$ being in the 0 basis state while the lower right block corresponds to $qubit_2$ in the 1 basis state. Note that if there were additional higher-order system qubits, the matrix *H(G)* would still be block diagonal but it would have a number of blocks exponential in the number of qubits not being transformed. Whenever a gate is operating on the low order qubits of a system state we will say that the circuit and the corresponding system Hamiltonian is in standard form. Our next code fragment simulates the application of a gate in standard form for a single time step:

```
ApplyGate1Step(Rho, H, dT)
...
LocalVariables:
...
GB    the number of rows (columns) in the basic gate Hamiltonian G
// The outer 2 loops iterate over the blocks of the block
// diagonal matrix H(G).  Note since all blocks are the
// same we only need the smaller matrix G.
for (jblk = 0; jblk++GB; kblk < QB}
      for (kblk = 0; kblk++GB; bklk < QB}
// The next three loops are the standard matrix multiply
// on blocks of Rho by G to compute T1 <- (G*Rho - Rho*G)
```

```
            for (j = 0; j++; j < GB)
                  for (k = 0; k++; k < GB)
                        T1 = 0
                        for (l =  0; l++; l < GS)
                              T1 += G[j,l]*Rho[l+jblk,k+kblk]
                              T1 -= Rho[j+jblk,l+kblk]*G[l,k]
                              Ts1[j+jblk,k+kblk] = f1*T1
// We use the same computation for the second order term
// noting that the block updates are independent
            for (j = 0; j++; j < GB)
                  for (k = 0; k++; k < GB)
                        T2 = 0
                        for (l =  0; l++; l < GS)
                              T2 += G[j,l]*Ts1[l+jblk,k+kblk]
                              T2 -= T1[j+jblk,l+kblk]*G[l,k]
                              Ts2[j+jblk,k+kblk] = f2*T2
// Finally, we combine the terms of the series, again
// the result blocks are independent.
            for (j = 0; j++; j < GB)
                  for (k = 0; k++; k < GB)
                        Rho[j+jblk, k+kblk] += Ts1[j+jblk, k+kblk]
                        Rho[j+jblk, k+kblk] -= Ts2[j+jblk, k+kblk]
```

Figure 13: Standard No Decoherence Matrix Simulation

The single large $2^N x 2^N$ matrix multiplication is replaced by N-g small $2^g x 2^g$ matrix multiplications. To see the impact this change has on performance, notice that the outer two loops cause the update section to be executed $2^{2(N-g)}$ times while the update section performs $O(2^{3g})$ operations. This leads to a computation requiring $O(2^{2N+g})$. Since g tends to be small (1, 2, or 3 usually) while N, which depends on the system, is the total number of qubits, we have $2N+g<<3N$. Thus, the new code offers a significant improvement over the previous version. Also, with the optimized multiplication, the large $2^N x 2^N$ unitary operator never needs to be constructed, one $2^g x 2^g$ block is sufficient. This process is illustrated in Figure 14. The gate Hamiltonian multiplies each of the individual blocks of the density matrix.

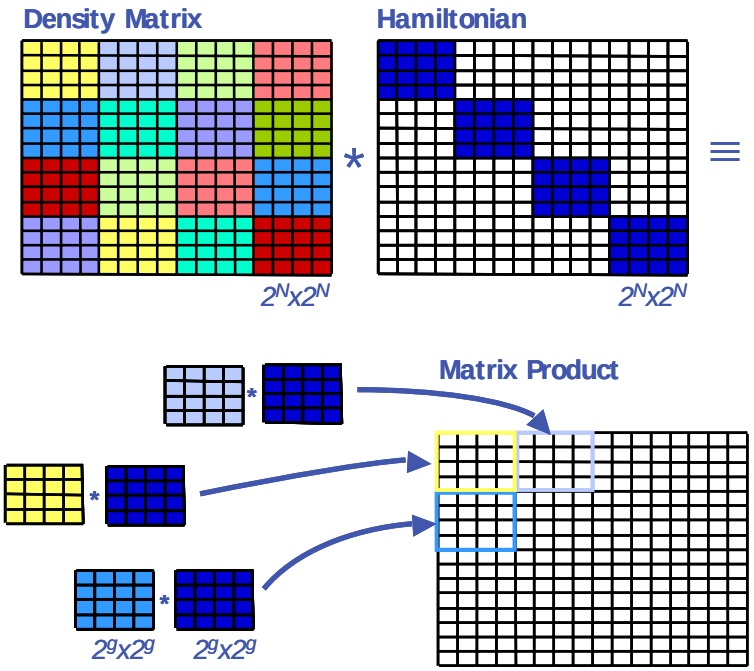

Figure 14: Canonical Density Matrix Multiplication, *N* = 4, *g* = 2

Unfortunately not all gates appear in standard position. (22) shows the same system gate as (21) but applied to different qubits, resulting in nonstandard form. The example on the left is being applied to qubits 0 and 2 while the one on the right is being applied to qubits 1 and 2.

$$H_{0,2} = \begin{pmatrix} a & b & 0 & 0 & c & d & 0 & 0 \\ e & f & 0 & 0 & g & h & 0 & 0 \\ 0 & 0 & a & b & 0 & 0 & c & d \\ 0 & 0 & e & f & 0 & 0 & g & h \\ i & j & 0 & 0 & k & l & 0 & 0 \\ m & n & 0 & 0 & o & p & 0 & 0 \\ 0 & 0 & m & n & 0 & 0 & o & p \\ 0 & 0 & m & n & 0 & 0 & o & p \end{pmatrix} \qquad H_{1,2} = \begin{pmatrix} a & 0 & b & 0 & c & 0 & d & 0 \\ 0 & a & 0 & b & 0 & c & 0 & d \\ e & 0 & f & 0 & g & 0 & h & 0 \\ 0 & e & 0 & f & 0 & g & 0 & h \\ i & 0 & j & 0 & k & 0 & l & 0 \\ 0 & i & 0 & j & 0 & k & 0 & l \\ m & 0 & n & 0 & o & 0 & p & 0 \\ 0 & m & 0 & n & 0 & o & 0 & p \end{pmatrix} \quad (22)$$

By inverting the transformations that produced these Hamiltonians from the standard gate definition, we can reduce the general problem to the specific problem of simulating a gate in standard position, producing the simplified block diagonal matrix multiplication implemented in Figure 13. The desired transformation mapping the gate into standard position can be thought of as a simple renaming of the system qubits. This renaming leads to a permutation of the rows and columns of the density matrix.

Code to generate the permutation of the row and column indices required to bring the algorithm and density matrix into standard form is presented in Figure 15. Recall that each index represents a computational basis state, one in which each of the qubits has a specific value, either 0 or 1. The index corresponding to a computational basis state is the binary number resulting from concatenating the values of the system's qubits in a fixed order, which we will call the input order. Standard form requires that the bits being acted upon by a gate be in the lowest order positions. Thus, for each index, we can compute its standard equivalent by permuting the system bits to bring the gate bits into the low order positions. The following pseudo-code achieves the required transformation of the indices. It produces a permutation which, for each standard position, *i*, identifies the corresponding index in input order. All that remains is to permute the rows and columns of the density matrix, then using this result, compute the Taylor series for *n* time steps, and then permute the rows and columns of the density matrix back to input order using the inverse of the permutation.

```
IndexPermutation(N, g, GateIn)
N           The number of qubits in the system
g           The number of qubits input to the gate
GateIn      A list of the g qubits the gate will be applied to
Standard    The permutation which, when applied to the rows and columns
            of the density matrix, will bring it into standard form.
            Note that P is a permutation of the integers 0..2^N - 1
// This is simply achieved.  For each index in P from
// 0 .. 2^N we permute its bits to move the bits in
// positions identified in GateIn to the low order
// positions.  The remaining bits are slid to the high
// order end to squeeze out the holes left by them.  The
// result is an index permuted to correspond to moving the
// gate into standard form.
      spaces = sort(GateIn, 'decreasing')
      for (i = 0; i++; i < N)
            newi = i
            for (k = 0; k++; k < N)
                  bit = getbits(i, k)
                  setbits(newi, k, getbits(i, k))
            for (k = 1; k++; k < N && spaces[k] >= N)
                  frag = getbits(i, (spaces[k]..spaces[k-1])
                  setbits(newi, (spaces[k]+k..spaces[k-1]+k), frag)
                  frag = getbits(i, (spaces[k]..spaces[k-1])
                  setbits(newi, (spaces[k]+k..spaces[k-1]+k), frag)
      Standard[i] = newi
```

Figure 15: Density Matrix Permutation Algorithm

To visualize the result of the permutation, Figure 16 shows the beginning portion of a 7 qubit Shor's algorithm. Figure 17 shows the same algorithm if the density matrix is permuted before each gate operation, resulting in the appearance that the gates are all in standard form.

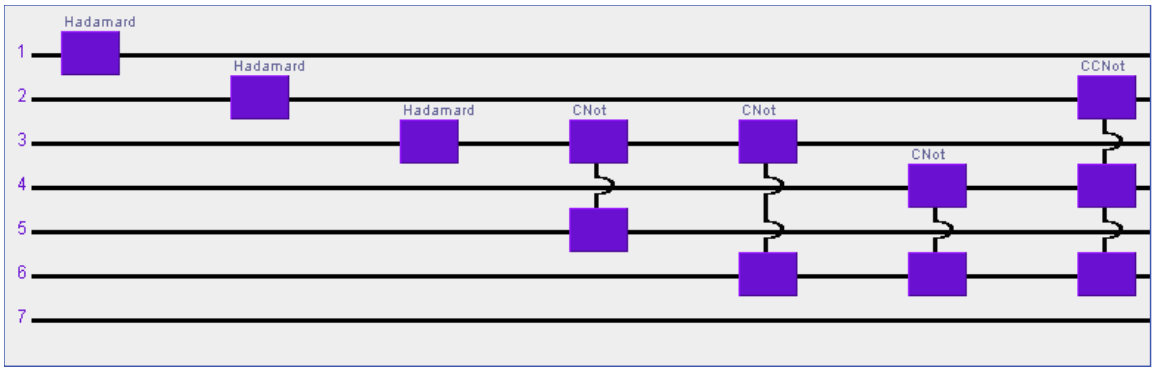

Figure 16: First Portion of 7 Qubit Shor's Algorithm

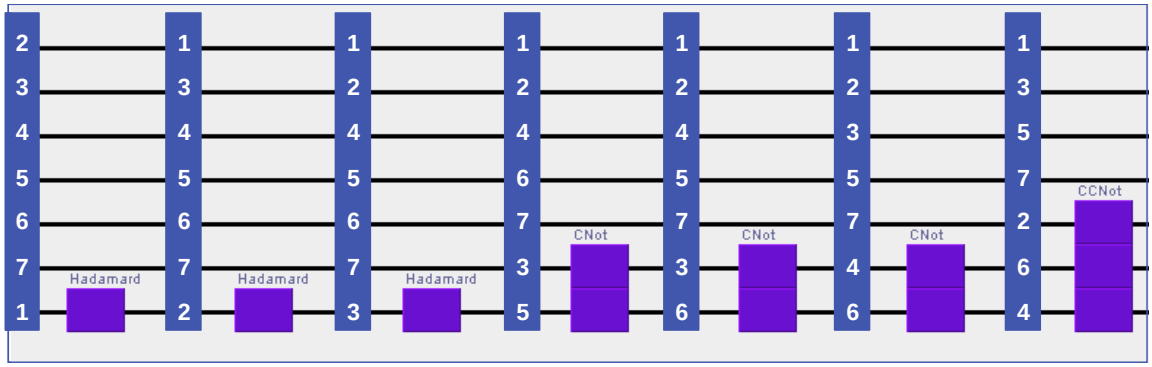

Figure 17: First Portion of 7 Qubit Shor's Algorithm with Permutation

Finally we must include the effects of decoherence. We address phase damping decoherence in this effort. To account for phase damping we must evaluate:

$$\sum_{j=1}^{N}(2V_j X V_j^H - V_j^H V_j X - X V_j^H V_j) \tag{23}$$

as part of computing the terms $\Omega\rho$ and $\Omega^2\rho$ of the Taylor series. First we will look at the form of V_j. Recall that for a single qubit the phase damping decoherence operator is expressed as:

$$V = \begin{bmatrix} 0 & 0 \\ 0 & b \end{bmatrix} = b\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} = b\hat{V}. \tag{24}$$

Again, using the example of a 3-qubit system, if we expand the single qubit definition we get the following:

$$b\hat{V}_0 = b\begin{pmatrix} 0 & & & & & & & \\ & 1 & & & & & & \\ & & 0 & & & 0 & & \\ & & & 1 & & & & \\ & & & & 0 & & & \\ & & 0 & & & 1 & & \\ & & & & & & 0 & \\ & & & & & & & 1 \end{pmatrix} \quad b\hat{V}_1 = b\begin{pmatrix} 0 & & & & & & & \\ & 0 & & & & & & \\ & & 1 & & & 0 & & \\ & & & 1 & & & & \\ & & & & 0 & & & \\ & & 0 & & & 0 & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{pmatrix}$$

$$b\hat{V}_2 = b\begin{pmatrix} 0 & & & & & & & \\ & 0 & & & & & & \\ & & 0 & & & 0 & & \\ & & & 0 & & & & \\ & & & & 1 & & & \\ & & 0 & & & 1 & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{pmatrix}. \tag{25}$$

As a consequence of the structure of the matrices it can be shown that:

$$V_i V_i^{\dagger} = bb^{\dagger}\hat{V}_i\hat{V}_i^{\dagger} = B\hat{V}_i \ . \tag{26}$$

Equation (26) allows us to simplify (23) yielding:

$$B\sum_{j=1}^{N}(2\hat{V}_j X\hat{V}_j - \hat{V}_j X - X\hat{V}_j)\,. \tag{27}$$

Since the $\hat{V}_i$ are diagonal binary matrices they possess properties that allow further algebraic simplification of (27) before it is included in the simulator code. The product $\hat{V}X$ has rows of 0 where the corresponding diagonal element of *V* is 0, otherwise it has the corresponding row of X in rows where the diagonal element of $\hat{V}$ is 1, thus acting like a row selector. Similarly, multiplication from the right selects columns.

At this point it would be useful to review an alternative form of the matrix product, $\otimes$, called the Hadamard product.

$$(A \otimes B)_{i,j} = a_{i,j} \times b_{i,j} \tag{28}$$

Notice that $\otimes$ is commutative and the matrix $\hat{1}$, a matrix of all 1's, serves as an identity. Thus we can further rewrite (27):

$$
\begin{aligned}
& B\sum_{j=1}^{N}(2\hat{V}_j\hat{1}\otimes X\otimes\hat{1}\hat{V}_j - \hat{V}_j\hat{1}\otimes X - X\otimes\hat{1}\hat{V}_j) \\
& = B\sum_{j=1}^{N}(2S_j\otimes X\otimes S_j^H - S_j\otimes X - X\otimes S_j^H) \\
& = X\otimes B\sum_{j=1}^{N}(2S_j\otimes S_j^H - S_j - S_j^H) \\
& = X\otimes\hat{W} \\
& \textit{where}\quad S_j = \hat{V}_j\hat{1}.
\end{aligned}
\tag{29}
$$

Notice that $\hat{W}$ only depends on the number of qubits. The following code fragment includes phase decoherence effects based on (29).

```
W     a matrix which is a precomputed function of user input
...
// The next three loops are the standard matrix multiply
// on blocks of Rho by G to compute T1 <- (G*Rho - Rho*G)
      for (j = 0; j++; j < GB)
            for (k = 0; k++; k < GB)
                  T1 = 0
                  for (l =  0; l++; l < GS)
                        T1 += G[j,l]*Rho[l+jblk,k+kblk]
                        T1 -= Rho[j+jblk,l+kblk]*G[l,k]
                        T1 += Rho[j+jblk,l+kblk]*W[j+jblk,l+kblk]
                        Ts1[j+jblk,k+kblk] = f1*T1
// We use the same computation for the second order term
// noting that the block updates are independent
      for (j = 0; j++; j < GB)
            for (k = 0; k++; k < GB)
                  T2 = 0
                  for (l =  0; l++; l < GS)
                        T2 += G[j,l]*Ts1[l+jblk,k+kblk]
                        T2 -= T1[j+jblk,l+kblk]*G[l,k]
                        T2 += Ts1[j+jblk,l+kblk]*W[j+jblk,l+kblk]
                        Ts2[j+jblk,k+kblk] = f2*T
```

Figure 18: Phase Decoherence Matrix Simulation Algorithm

Finally, we need to compute $\hat{W}$.

$$
\begin{aligned}
\hat{W} &= X \otimes B\sum_{j=1}^{N}(2S_j \otimes S_j^H - S_j - S_j^H) \\
&= B(\sum_{j=1}^{N} 2S_j \otimes S_j^H - \sum_{j=1}^{N} S_j - \sum_{j=1}^{N} S_j^H) \\
&= B(2\sum_{j=1}^{N} V_j - \sum_{j=1}^{N} S_j - \sum_{j=1}^{N} S_j^H) \\
&= B(2\sum_{j=1}^{N} V_j - \sum_{j=1}^{N} V_j \otimes \hat{1} - \sum_{j=1}^{N} \hat{1} \otimes V_j^H) \\
&= B(2(\sum_{j=1}^{N} V_j) - (\sum_{j=1}^{N} V_j) \otimes \hat{1} - \hat{1} \otimes (\sum_{j=1}^{N} V_j)^H).
\end{aligned}
\tag{30}
$$

Going back to the definition of V_j we see that $(V_j)_{i,i} = Bits(i)_j$, the j^{th} bit of the binary representation of i. Thus the i^{th} component of the first sum is the number of ones in the binary representation of j.

4.1.3 Optimization Results

These enhancements have resulted in substantial performance improvements in the execution of Quantum eXpress. Table 2 shows the pre-optimization simulation times accompanied by the post-optimization simulation times.

Table 2: Post-Optimization Simulation Times

	No Decoherence	Original Decoherence	Optimized Decoherence
3 Qubit	2.1s	5.2s	1.01s
5 Qubit	4.6s	720.6s (12m 0.6s)	20.77s
7 Qubit	88s (1m 28s)	155191.4s (43h 6m 31s)	758s (12m 38s)

We were able to take advantage of the unique structure of the system Hamiltonian matrices to reorganize their bits to make the matrix appear block diagonal. By permuting the same bits of the density matrix, we produced the equivalent matrix multiplication. We were then able to replace the single large matrix multiplication with many smaller multiplications, substantially reducing the number of calculations and memory overhead required.

Table 3 provides the percent improvement from the original decoherence to the optimized decoherence, and Figure 19 plots those results.

Table 3: Decoherence Improvement from Original Implementation

	Improvement (%)
3 Qubit	80.58%
5 Qubit	97.12%
7 Qubit	99.51%

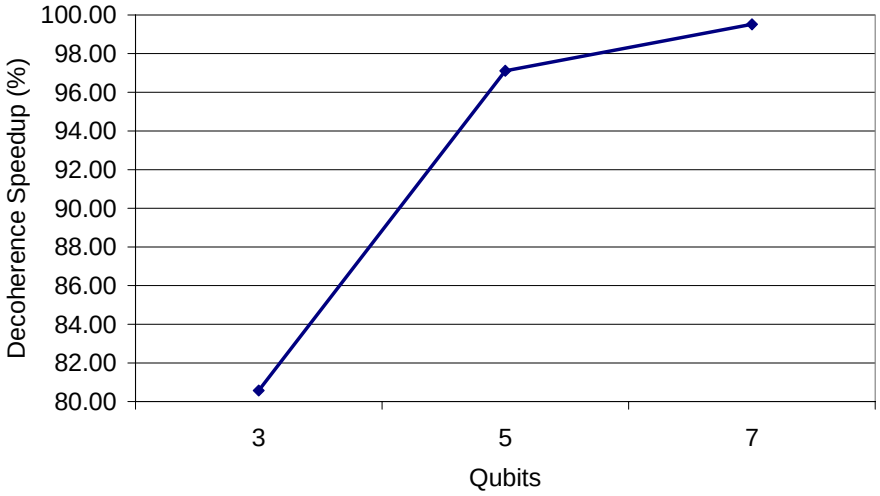

Figure 19: Simulation Performance Improvement

4.2 Port to High Performance Computing Cluster

The new algorithm for gate application significantly improved the performance of Quantum eXpress. Beyond those optimizations, porting QX to a high performance-computing environment could also significantly reduce the amount of time required to execute a simulation. It could also increase the number of qubits able to be simulated, enabling a more thorough investigation of error correction schemes. This would allow researchers to better quantify the amount of error correction required to develop a large-scale quantum computer with high fidelity. A byproduct of the optimized implementation described above is an improved locality of reference resulting from smaller matrix multiplications that naturally lends itself to a distributed implementation. Each of the small 2^gx2^g matrix multiplications in the optimized simulation is independent. Thus, all of them can occur in parallel. Therefore, in theory, doubling the number of CPU's in the execution could reduce the computational time by half (neglecting the processing overhead on the head node and the cost of distributing the data across the network).

Quantum eXpress is divided into two components—a graphical user interface (GUI) for designing quantum algorithms and specifying initial states, and a back-end simulator engine responsible for evaluating the algorithms. The GUI can invoke a local instance of the simulator engine or it can connect to a remote server via the Simple Object Access Protocol (SOAP) to execute the simulator engine running on a shared server, as shown in Figure 20.

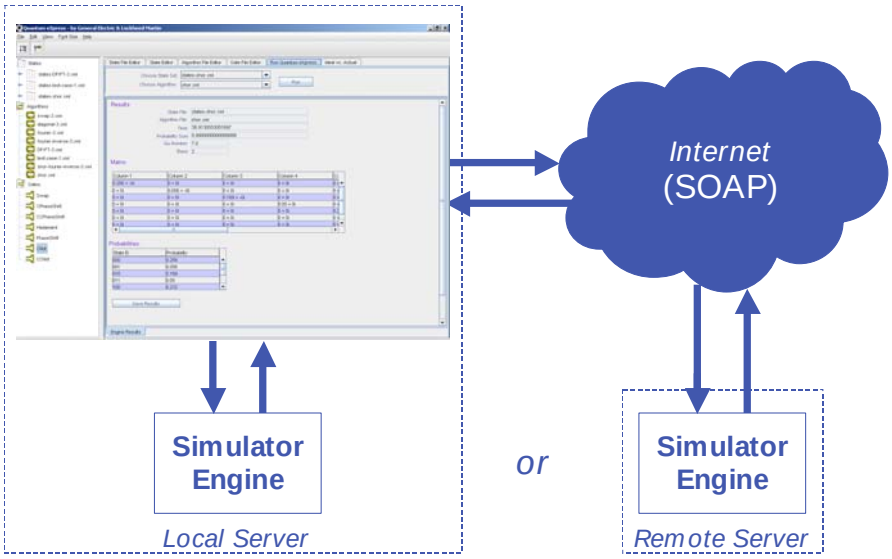

Figure 20: Quantum eXpress Connected to Either a Local or Remote Server

This division of labor allows us to concentrate our performance enhancement efforts on the back end simulator while the GUI remains unchanged on a researcher's desktop. The parallelized version is embodied in an alternative back end that executes on a cluster. The head node of the cluster is responsible for managing the density matrix, initializing the Hamiltonian, and dividing and distributing the density matrix and Hamiltonian to the cluster nodes to execute the parallel evaluations.

The distributed implementation utilizes Java's Remote Method Invocation (RMI) [9] capabilities to communicate with and pass data to the various nodes. A Web server is also required, to provide the nodes with access to the required QX code to perform the simulations. In our implementation, the Web server runs on the head node. It is started using the script shown in Figure 21.

```
#!/bin/tcsh

echo Starting webserver on port 8080
java -jar lib/tools.jar -dir . -verbose -port 8080
```

Figure 21: Script to Start Web Server on Head Node (startWebServer.sh)

The individual cluster nodes each connect with the Web server via RMI to download the simulator code. Each node in the cluster thus must be made aware of the head node's Web server URL, and is started with the script shown in Figure 22.

```
#!/bin/tcsh

# set properties
# Web URL
```

```
setenv WebURL http://master.afrl.mil:8080/

# RMI URL
setenv RMIport 1099
setenv RMIURL rmi://hostname0.afrl.mil:$RMIport/qxNode

# start RMI registry
echo Starting RMI registry on port $RMIport
rmiregistry $RMIport &
sleep 2

# start Node
echo Starting QX node with RMI URL $RMIURL
java -classpath .:./lib/quantumexpress.jar::./lib/jdom.jar
-Djava.rmi.server.codebase=$WebURL -Djava.security.policy=
security/policy.all com.quantum.engine.QuantumClusterServer $RMIURL
```

Figure 22: Script to Start a Cluster Node (startNode.sh)

To know how to communicate with each of the cluster nodes, an XML configuration file, accessible by the head node, lists all of the available nodes. Figure 23 shows an example configuration file.

```
<nodes>
  <distributed> true </distributed>
  <node> rmi://hostname0.afrl.mil:1099/qxNode </node>
  <node> rmi://hostname1.afrl.mil:1099/qxNode </node>
  <node> rmi://hostname2.afrl.mil:1099/qxNode </node>
  <node> rmi://hostname3.afrl.mil:1099/qxNode </node>
  ...
</nodes>
```

Figure 23: XML Configuration for Head Node to Find Cluster Nodes (nodes.xml)

Figure 24 shows how the cluster nodes and Web server interact, and how the QX client runs on the head node to distribute work to each of the cluster nodes.

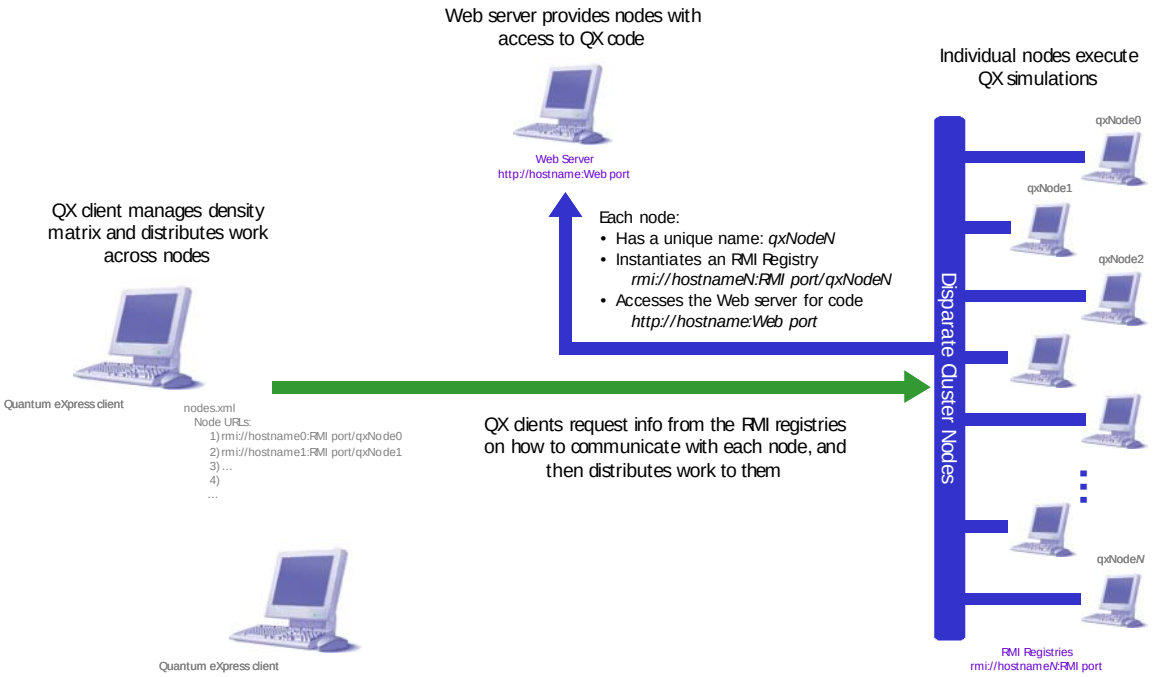

Figure 24: Cluster Node Communication

4.2.1 Density Matrix Distribution

Once all of the nodes are started and a simulation has begun, the head node first determines the Hamiltonian that must be used for each gate. It then determines how many nodes in the cluster to use for the parallel simulation. The maximum number of nodes that can be used is dependent on the number of qubits N in the density matrix, and the number of qubits g being operated upon by the gate in question. The maximum number of usable nodes per gate is the number of 2^gx2^g matrices that the full 2^Nx2^N density matrix can be divided into:

$$2^{2(N-g)}. \tag{31}$$

Clearly, the maximum number of usable nodes is a power of 2. Assuming each node in the cluster is equivalent, there is no benefit to dividing the density matrix into uneven portions, so the head node divides the density matrix by powers of 2. First, the density matrix is divided into as many columns as possible, as long as the number of columns does not exceed the number of available nodes and the minimum column width is 2^g. An example of repeatedly dividing the density matrix into columns is shown in Figure 25.

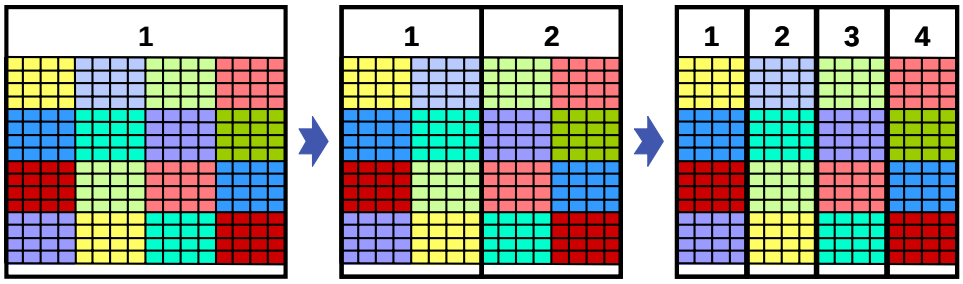

Figure 25: Density Matrix Column Division Example, *N* = 4, *g* = 2

If more nodes are available, the density matrix is further divided into rows. As expected, the minimum row width is 2^g. An example of further dividing the density matrix into rows is shown in Figure 26.

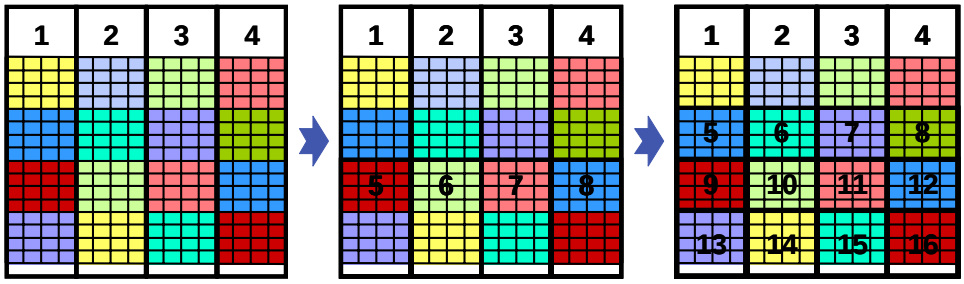

Figure 26: Density Matrix Row Division Example, *N* = 4, *g* = 2

Once the head node divides the density matrix, it communicates with each node in the cluster to distribute a portion of the density matrix and the Hamiltonian. Each cluster node then executes the simulation and passes the resulting density matrix back to the head node.

4.2.2 Distributed Simulation

We use the code for applying a gate in canonical position as the basis for partitioning the problem for distributed implementation. Notice that the outer two loops iterate over a set of blocks of ρ performing matrix multiplications. The multiplications have no data dependencies thus they can be treated as independent threads. Although we did not explicitly treat the multiple evaluation of the Taylor series expansion to simulate multiple time steps, it should be clear that the independence of the data in the threads could be used to bring that iteration inside the thread.

```
forall (jblk = 0; jblk+=GB; jblk < QB)
      forall (kblk = 0; kblk+=GB; kblk < QB)
        Thread(Rho, G, W, jblk, kblk, GB)
// The next three loops are the standard matrix multiply
// on blocks of Rho by G to compute T1 <- (G*Rho - Rho*G)
            for (j = 0; j++; j < GB)
                  for (k = 0; k++; k < GB)
                        T1 = 0
```

```
                        for (l =  0; l++; l < GS)
                              T1 += G[j,l]*Rho[l+jblk,k+kblk]
                              T1 -= Rho[j+jblk,l+kblk]*G[l,k]
                              T1 += Rho[j+jblk,l+kblk]*W[j+jblk,l+kblk]
                              Ts1[j+jblk,k+kblk] = f1*T1
// We use the same computation for the second order term
// noting that the block updates are independent
            for (j = 0; j++; j < GB)
                  for (k = 0; k++; k < GB)
                        T2 = 0
                        for (l =  0; l++; l < GS)
                              T2 += G[j,l]*Ts1[l+jblk,k+kblk]
                              T2 -= T1[j+jblk,l+kblk]*G[l,k]
                              T2 += Ts1[j+jblk,l+kblk]*W[j+jblk,l+kblk]
                              Ts2[j+jblk,k+kblk] = f2*T
```

Figure 27: Distributed Simulation Algorithm

As expected, the distributed simulation also significantly improved the performance of the simulator. Table 4 shows the results of distributing the three algorithm simulations across a cluster, with different numbers of nodes available. The number of nodes used at each step are shown, along with the resulting simulation times. The optimal number of nodes used and simulation times are in bold.

Table 4: Distributed Simulation Times

	3 Qubits		5 Qubits		7 Qubits	
Nodes Available	**Time (sec)**	**Nodes Used**	**Time (sec)**	**Nodes Used**	**Time (sec)**	**Nodes Used**
0	1.01	0	20.77	0	758	0
1	1.13	1	21.43	1	699	1
2	0.95	2	11.36	2	364	2
4	**0.92**	**4**	6.46	4	199	4
8	0.97	4	4.16	8	124	8
16	0.98	4	**3.36**	**16**	**95**	**16**

Figure 28 shows a plot of the distributed simulation times from Table 4.

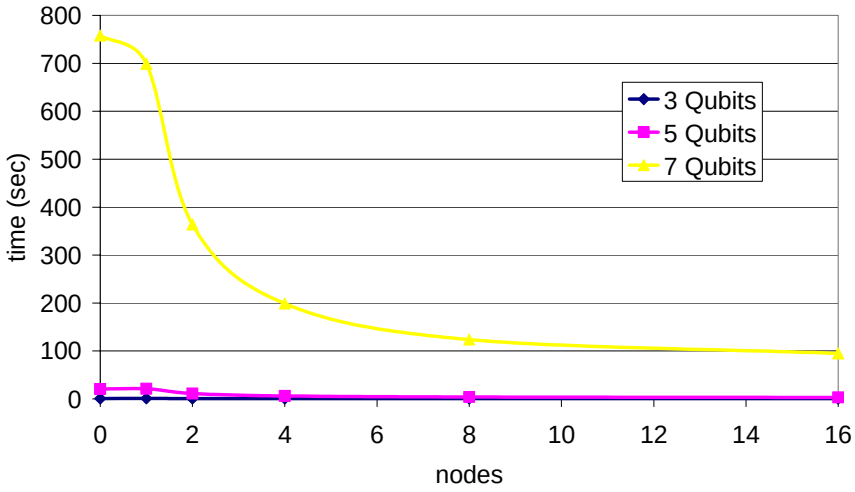

Figure 28: Distributed Simulation Times vs. Number of Nodes

Table 5 shows how increasing the number of cluster nodes at each row in Table 4 improves the performance of the simulation as compared to the non-distributed (zero nodes) simulation. If we assume the head node causes no overhead in the simulation, and that there is no overhead due to distributing data across the network to the cluster nodes, than we can calculate the best possible speed-up, as is shown under the 'ideal' column in Table 5.

Table 5: Distributed Simulation Time Improvement Compared to Non-Distributed Simulation

Nodes	Ideal	3 Qubits	5 Qubits	7 Qubits
0	0.00%	0.00%	0.00%	0.00%
1	0.00%	-11.88%	-3.18%	7.78%
2	50.00%	5.94%	45.31%	51.98%
4	75.00%	**8.91%**	68.90%	73.75%
8	87.50%	3.96%	79.97%	83.64%
16	93.75%	2.97%	**83.82%**	**87.47%**

In comparing the ideal and actual speed-ups, we expect the actual results to be slightly worse. This is due to overhead in both the computations performed on the head node as well as the network time required for distributing the components of the density matrix. For example, for the single node case the ideal speed-up is zero, but we see that for the 3 qubit and 5 qubit examples the actual speed-up is negative. This is expected because time is spent distributing the density matrix to that node, but no speed-up is incurred using a single node for the calculations. Similarly, since the 3 qubit example can use at most 4 nodes, we see the maximum speed-up occurs with 4 nodes. Adding more nodes adds only additional processing overhead, reducing the performance improvement.

An interesting result found in Table 5 is that for the single node example, the 7 qubit test case produces a positive speed-up. This is in contrast to the zero-percent ideal improvement, and negative actual speed-up anticipated. Similarly, for the experiment with two nodes we again see that the 7 qubit test case has a slightly higher improvement than expected under ideal circumstances. Dozens of experiments were run to verify that both of these results were indeed produced by the optimized implementation of Quantum eXpress. Since they conflict with expectations, further investigation is required to find a valid explanation. Otherwise, the remaining results all met with our expectations, indicating that the optimization was quite successful. A plot of the percent-improvements shown in Table 5 can be found in Figure 29.

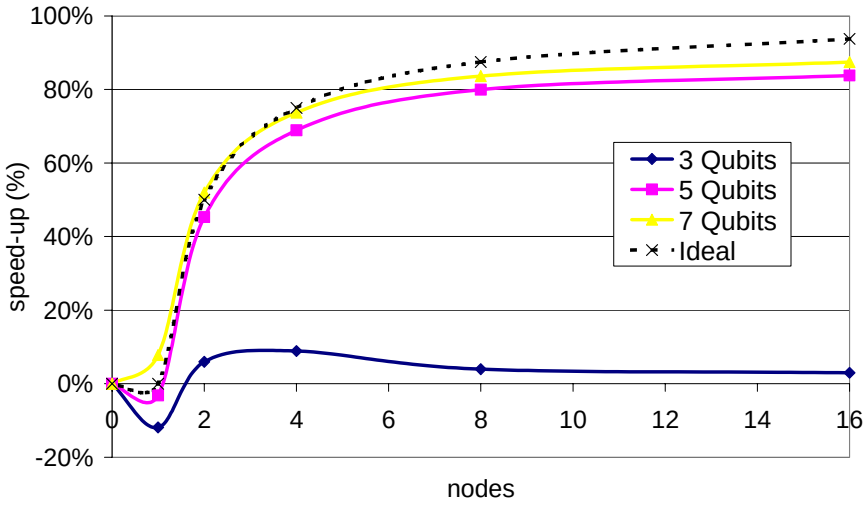

Figure 29: Distributed Simulation Improvements Compared to Non-Distributed Implementation

Table 6 shows how the optimal simulation times from Table 4 compare to the original implementation and the optimized simulation prior to distribution across the cluster.

Table 6: Distributed Simulation Time Comparison

	Original Decoherence	Optimized Decoherence	Optimized & Distributed Decoherence
3 Qubit	5.2s	1.01s	0.92s
5 Qubit	720.6s (12m 0.6s)	20.77s	3.36s
7 Qubit	155191.4s (43h 6m 31s)	758s (12m 38s)	95s (1m 35s)

Table 7 shows the percent improvement in simulation time from the original implementation to the optimized simulation, and from the original implementation to the optimized and distributed simulation.

Table 7: Decoherence Improvement from Original Implementation

	Optimized Improvement (%)	Optimized & Distributed Improvement (%)
3 Qubit	80.58%	82.31%
5 Qubit	97.12%	99.53%
7 Qubit	99.51%	99.94%

Figure 30 plots the data from Table 7, visualizing the performance improvement experienced as a result of the distributed simulation.

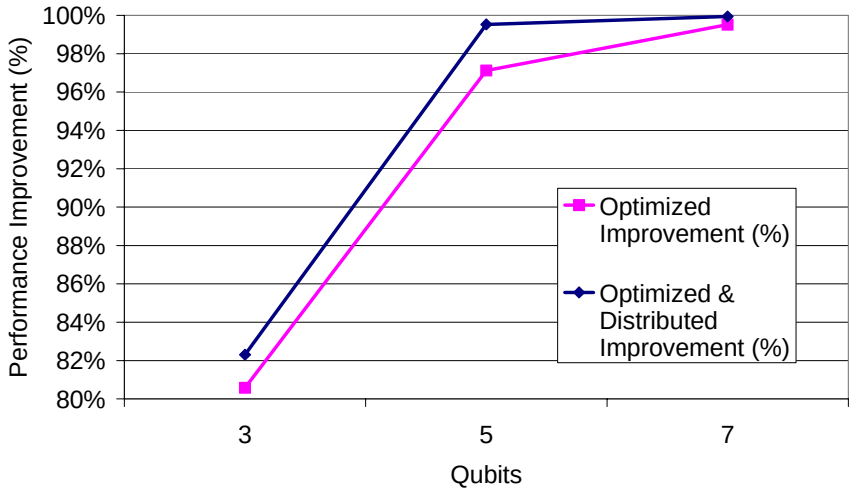

Figure 30: Decoherence Simulation Performance Improvement from Original Implementation

4.3 Ideal vs. Actual Density Matrix Comparisons

As described, QX is capable of simulating a quantum algorithm with both device and decoherence errors simulated. It is also capable of simulating algorithms under ideal (error-free) circumstances. During an algorithm's execution, it would be valuable to simulate, in parallel, the algorithm's execution in both ideal and actual circumstances and visualize the difference in the density matrix at each step in the algorithm. This would enable researchers to see exactly how errors affect the state of the quantum system and how they propagate over time.

The simulator was therefore updated to simulate and calculate the difference between the ideal (no errors, no decoherence) density matrix and the actual (error/decoherence-based) density matrix after each gate. If the simulator is distributed over a cluster, the cluster is used for the actual simulation step and then the ideal simulation step. These are not run in parallel because the actual simulation step requires significantly more time, and therefore splitting the cluster in half and simulating the two steps in parallel would be notably slower than running the ideal step after the actual. The difference between the two matrices

is stored after each step, along with the Frobenius Norm of the difference between those two matrices.

The Frobenius Norm is a generalized Euclidian norm that provides the 'distance' between two matrices [10]. The norm involves calculating the square root of the sum of the absolute squares of a matrix element:

$$\|F\| = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}\left|a_{i,j}\right|^2} \; . \tag{32}$$

4.3.1 Visualizing Matrix Differences

Quantum eXpress has also been enhanced to visualize the difference after each gate, along with the Frobenius Norm. Heatmaps are used to visualize the cells in the matrix that have non-zero elements, with different colors to represent various numeric cutoffs. Cells are colored to indicate which have numerical values greater than user-defined thresholds. This allows users to focus their attention on the areas with higher error values that desirable, without needing to examine the entire matrix. Three colors are used, and they have default values associated. Those colors and default cut-offs are shown in Table 8. The color cutoff values can be changed directly in the QX user interface.

Table 8: Ideal vs. Actual Heatmap Default Cutoffs

Color		Default Cutoff
Yellow		10^{-5}
Orange		10^{-3}
Red		10^{-1}

Example visualizations are shown in Figure 31, for a system where $N = 3$.

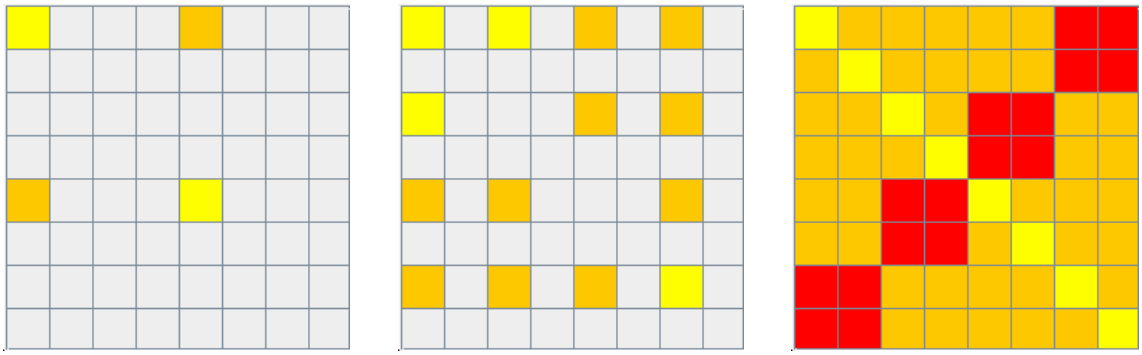

Figure 31: Example Matrix Difference Visualizations, $N = 3$

The simulator does not calculate the ideal vs. actual density matrix differences by default. The following parameter must be added to the states.xml configuration file, and set to 'true' in order for the differences to be calculated.

```
<compIdealVSActual> true </compIdealVSActual>
```

Figure 32: Ideal vs. Actual Matrix Calculation XML Parameter

4.4 Simulator Verification

4.4.1 Experimental Data Collection

The first step to verifying the simulator was to find relevant ion trap quantum computer experiments. We identified about 40 research groups working in this area. We found the groups through the Internet, textbooks, technical papers, and Rome Labs contacts. We determined that the following groups had the most advanced and relevant research programs: Innsbruck University, University of Michigan, National Institute of Standards and Technology (NIST), Oxford University, and Imperial College. Some of the other groups that we investigated were the following: University of Aarhus, MIT, Georgia Tech, Johannes-Gutenberg University, University of Washington, University of Illinois, Weizmann Institute of Science, and Penn State University.

We reviewed all the publications of the more advanced and relevant groups, and identified three candidate algorithms:

- Deutsch-Jozsa [11]
- Grover [12]
- Semi-classical quantum Fourier transform (QFT) [13]

After reviewing the published data, we identified three main project risks and developed a plan to address them. The first risk was that we needed to obtain complete and consistent experimental data. To avoid a critical dependency on a single data source, we decided it would be best to attempt to obtain data for each of the candidate algorithms. We planned to fill any gaps in the data that we obtained by making educated approximations based on the data we could find elsewhere in the literature. We also decided to use MATLAB to test the algorithms that we found from the experimentalists to verify that they produced the correct results in the absence of noise.

The second risk was that experimental decoherence data is difficult to obtain. It was clear from our literature review that the experimentalists had only a rudimentary understanding of decoherence sources in their experiments. We deferred decoherence analysis to future research. We similarly deferred testing any of the simulator's capabilities that would not be required for the experiments we modeled.

To model each algorithm, we required the following:

- the algorithm as implemented in the experiment
- gate Hamiltonians
- gate application times
- gate noise estimates
- decoherence data
- experimental results

Some of this data was available in the published papers. To obtain what wasn't readily available, we attended the "Workshop on Trapped Ion Quantum Computing" at NIST, February 21 – 24, 2006. We met experimentalists from each of the three relevant research groups and secured most of the data that we needed.

We confirmed with the experimentalists that they do not have sufficient understanding of decoherence in their experiments for us to model it. Their experiments are focused on producing working algorithms, and little research on analysis of noise sources had been done to this point. We also were unable to obtain the Hamiltonians and gate noise estimates for Michigan's Grover algorithm experiments. We filled in the gaps using the other experiments and other published papers from the Michigan group.

We tested each of the experimental algorithms in MATLAB and verified that they produce the correct results in the absence of noise.

4.4.2 Modeling Experiments in the Simulator

To model the experimental algorithms, we had to translate the experimental data into simulation designs and input the designs into Quantum eXpress as XML files. In the remainder of this section, we describe the purpose of each algorithm and its Quantum eXpress design.

Deutsch-Jozsa Algorithm

The purpose of the Deutsch-Jozsa algorithm as experimentally implemented is to determine whether a binary function f with a binary input is constant or balanced [11]. If it is constant, then it will produce either 0 or 1, regardless of its input. If it is balanced, then it will either reproduce its input, or produce the complement of its input (see Table 9).

Figure 33 shows the two-qubit Deutsch-Jozsa algorithm, which works as follows. The primary qubit "a" is initialized to |0>, and the secondary qubit "w" is initialized to |1>. Each qubit undergoes a rotation to put it in a superposition state. Then there is a unitary operator that operates on both qubits and depends on which case is being implemented (see Table 9). The effect of the operator is to perform addition modulo 2 of the secondary qubit and the function evaluated with the primary qubit. The qubits then undergo inverse rotations to prepare for measurement. Finally, the probability that the primary qubit is in state |1> is

measured. If the function is constant, then the probability should be zero, and if the function is balanced, then the probability should be unity.

Table 9: Constant and Balanced Functions

	Constant Functions		Balanced Functions	
	Case 1	**Case 2**	**Case 3**	**Case 4**
f(0)	0	1	0	1
f(1)	0	1	1	0

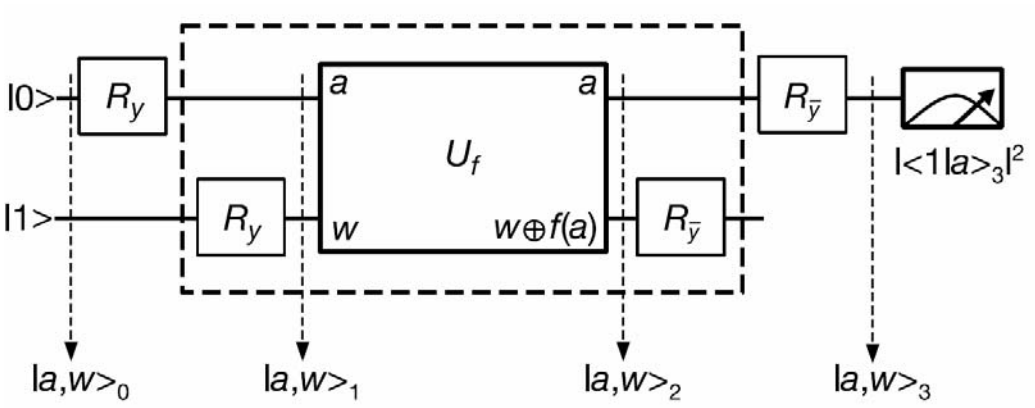

Figure 33: Deutsch-Jozsa Algorithm [11]

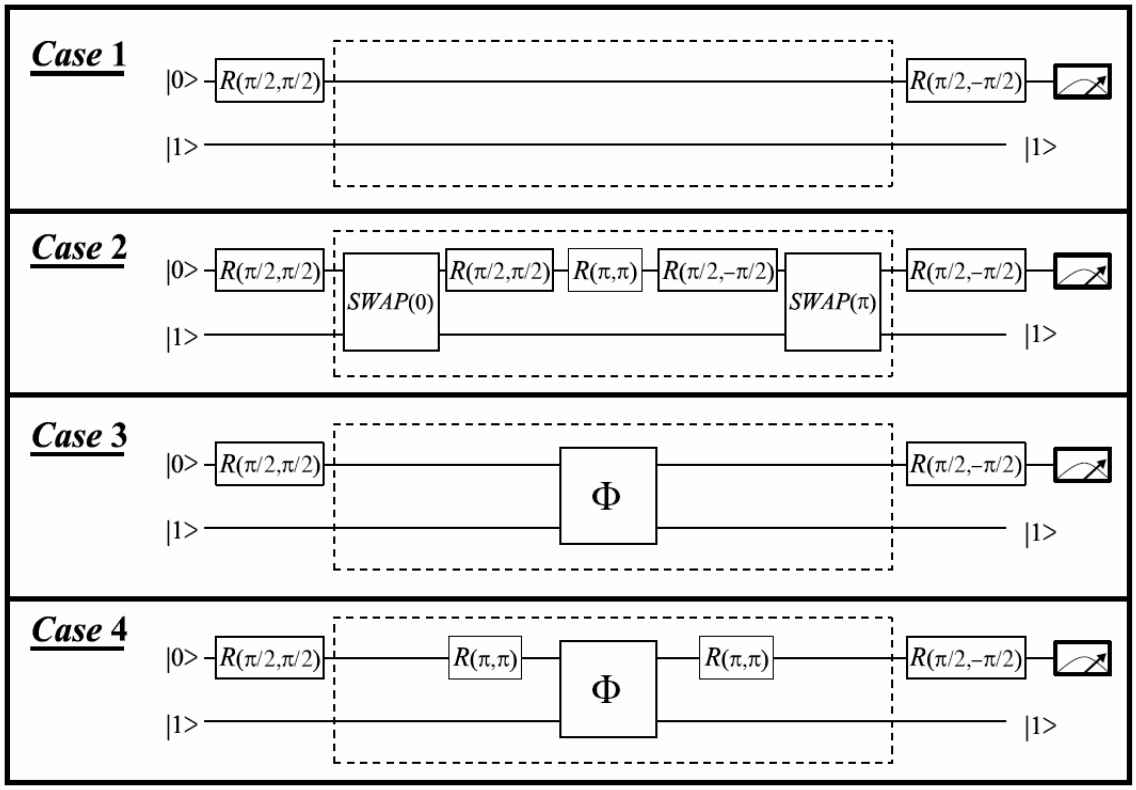

Figure 34: Experimental Implementation of the Deutsch-Jozsa Algorithm [11]

Note that the circuit is measuring a global property of the function f, which is the strength of quantum algorithms. The advantage of the quantum algorithm over a classical algorithm is that the quantum algorithm only requires a single evaluation of the function, while classically two evaluations are required.

Figure 34 shows the experimental implementation of the algorithm for each case. The circuit design for the simulator is the same as in the figure, except there is an additional working qubit that is initialized to |0>, runs across the top of each circuit, and is covered by the swap and phase (Φ) gates. The additional qubit is necessary because the experiment takes the system out of the two-qubit computational basis in the intermediate portion of the calculation, and the simulator cannot handle that without a third qubit.

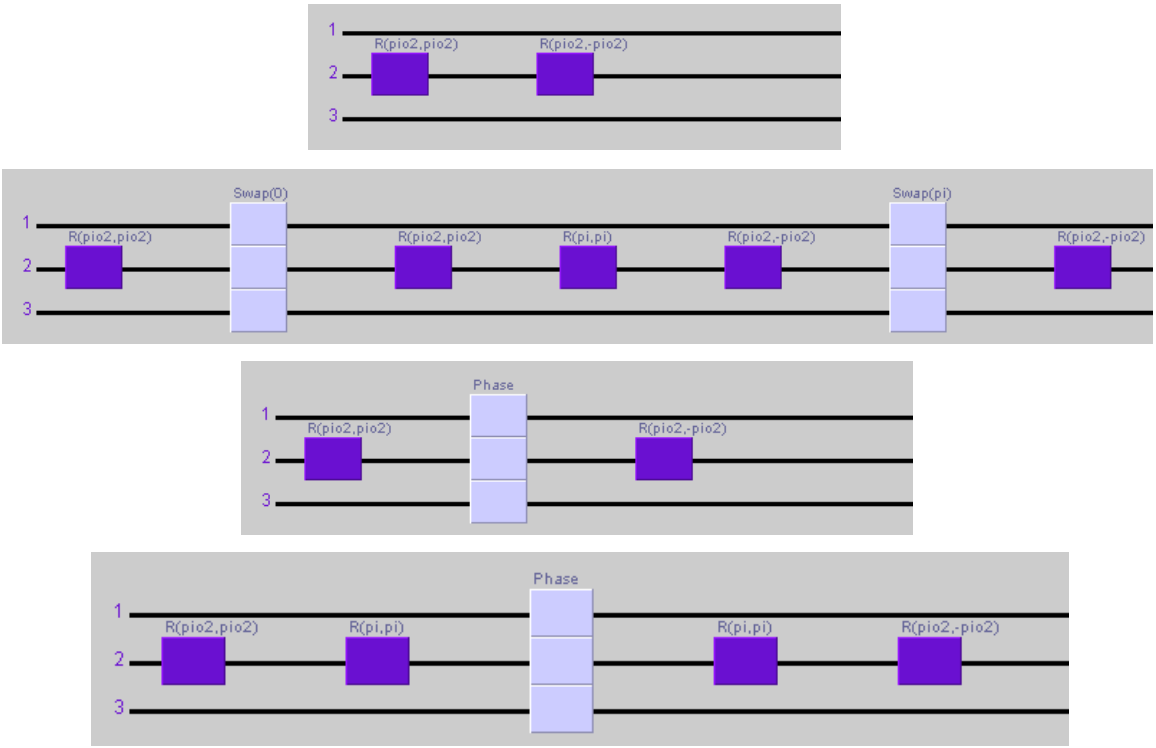

Figure 35: GUI View of the Deutsch-Jozsa Circuits

The rotation gates $R(\theta,\phi)$ are generated by applying the carrier Hamiltonian for a time t:

$$H^{C} = \frac{\Omega}{2}\begin{pmatrix} 0 & e^{-i\phi} \\ e^{i\phi} & 0 \end{pmatrix}, \tag{33}$$

where $t = \theta/\Omega$, and Ω is a parameter that represents the strength of the coupling between the ion and the laser that is used to manipulate it. Simulations

are independent of the value of Ω because the Hamiltonian and gate time always appear in the combination $H^{C} t$, and Ω cancels out of this product.

The swap and phase gates are defined as sequences of blue sideband rotations:

$$R_{\text{phase}} = R^{+}(\pi, \frac{\pi}{2})\, R^{+}(\frac{\pi}{\sqrt{2}}, 0)\; R^{+}(\pi, \frac{\pi}{2})\, R^{+}(\frac{\pi}{\sqrt{2}}, 0), \tag{34}$$

$$R_{\text{swap}}(\phi_0) = R^{+}(\frac{\pi}{\sqrt{2}}, \phi_0)\, R^{+}(\frac{2\pi}{\sqrt{2}}, \phi_0 + \phi_{\text{swap}})\; R^{+}(\frac{\pi}{\sqrt{2}}, \phi_0). \tag{35}$$

Table 10: Deutsch-Jozsa Simulator Results with No Noise

Case 1		Case 2	
State	**Probability**	**State**	**Probability**
|000>	0.0	|000>	2.3034916202E-16
|001>	0.9999999366	|001>	0.9999970832
|010>	0.0	|010>	2.3591297278E-16
|011>	0.0	|011>	1.3025527459E-12
|100>	0.0	|100>	5.5554475174E-35
|101>	0.0	|101>	0.0
|110>	0.0	|110>	5.5554475196E-35
|111>	0.0	|111>	2.3034916202E-16
Case 3		**Case 4**	
State	**Probability**	**State**	**Probability**
|000>	5.6249534896E-18	|000>	1.1842324339E-17
|001>	1.0063527674E-13	|001>	1.0041509916E-13
|010>	5.6249534873E-18	|010>	1.1842324344E-17
|011>	0.9999993025	|011>	0.9999982879
|100>	0.0	|100>	0.0
|101>	0.0	|101>	0.0
|110>	0.0	|110>	0.0
|111>	0.0	|111>	0.0

Note that gates are applied in opposite order of these rotation matrix multiplications, and

$$\phi_{\text{swap}} = \arccos(\cot^{2}(\frac{\pi}{\sqrt{2}})). \tag{36}$$

$R^{+}(\theta, \phi)$ is generated by applying the blue sideband Hamiltonian for a time t:

$$H^{\mathrm{B}} = \frac{\Omega}{2}\begin{pmatrix} 0 & 0 & 0 & 0 & \sqrt{2}e^{-i\phi} & 0 & 0 & 0 \\ 0 & 0 & e^{-i\phi} & 0 & 0 & 0 & 0 & 0 \\ 0 & e^{i\phi} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \sqrt{2}e^{i\phi} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad (37)$$

where $t = \theta / \Omega$, and Ω is an arbitrary parameter that represents the strength of the coupling between the ion and the laser that is used to manipulate it.

The simulator cannot handle Hamiltonians with parameters in their matrix elements, so separate gates had to be constructed for each set of angle arguments. Figure 35 shows what the circuits look like in the Quantum eXpress GUI. For each case, the initial state is |001>.

Table 10 shows the simulator's output for each case in the absence of gate noise and decoherence. Note that the probability that the primary qubit (the second of the three qubits) is in state |1> is very close to zero for cases 1 and 2, and very close to unity for cases 3 and 4. That is the expected result. The deviations from the ideal results are due to numerical approximations and round-off errors.

Grover Algorithm

The purpose of the Grover algorithm as experimentally implemented is to find a target state in a search space of four states [12]. Figure 36 is a schematic illustration of the general algorithm. The state is initialized to all zeros and then undergoes a Hadamard transformation that puts it in an equal superposition of all states (see Figure 36a). Then a portion of the circuit called the "oracle" marks the target state by flipping the sign of its amplitude in the superposition (see Figure 36b). Two additional Hadamard transformations with a phase gate between them then amplify the amplitude of the target state (see Figure 36c). Finally, the state is measured, and hopefully is the target state. To increase the probability of success, steps (b) and (c) can be repeated many times. In the special case of a 4-D search space, only one iteration is necessary to guarantee success.

Figure 37 shows the two-qubit Grover algorithm as implemented in the experiment. Both qubits are initialized to |0>, and then undergo rotations to put them in superposition states. Then the rotations (in the dark boxes) swap the target state (which is marked by choosing the appropriate angles α and β) and the |11> state. Next, the controlled Z gate portion of the circuit flips the sign of the |11> state. Then the rotations in the dark boxes swap the target state and

the |11> state again, and the portion of the circuit in dark gray amplifies the amplitude of the target state so its magnitude is unity.

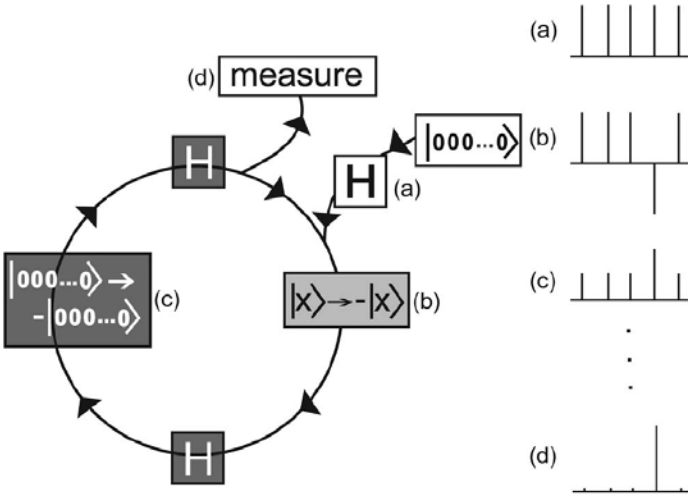

Figure 36: Grover Algorithm [12]

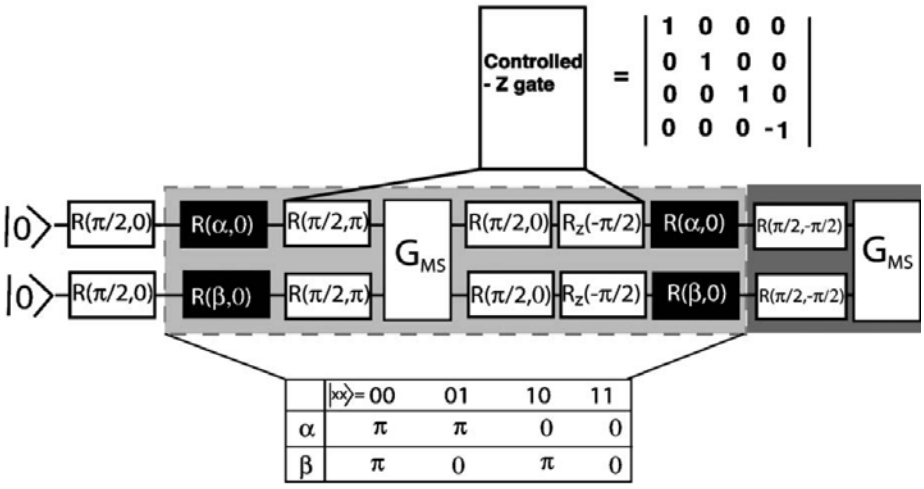

Figure 37: Experimental Implementation of the Grover Algorithm [12]

For Quantum eXpress, the circuit design is the same as in Figure 37. We approximated $R(\theta,\phi)$ to be the same as for the Deutsch-Jozsa circuits, and

$$R_Z(-\frac{\pi}{2}) = R(\frac{3\pi}{2},\frac{\pi}{2})\ R(\frac{3\pi}{2},0)\ R(\frac{3\pi}{2},-\frac{\pi}{2})\,. \tag{38}$$

G_{MS} is generated by applying the Mølmer-Sørensen Hamiltonian for a time t [14]:

$$H^{\mathrm{MS}} = \frac{\Omega}{2}\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \tag{39}$$

where $t = \pi/(2\Omega)$, and Ω is an arbitrary parameter that represents the strength of the coupling between the ion and the laser that is used to manipulate it.

Since the simulator cannot handle Hamiltonians with parameters in their matrix elements, separate gates again had to be constructed for each set of angle arguments. Figure 38 shows the circuit for the |00> target state in the Quantum eXpress GUI. The circuits for the other target states have the same structure. For each target state, the initial state is |00>.

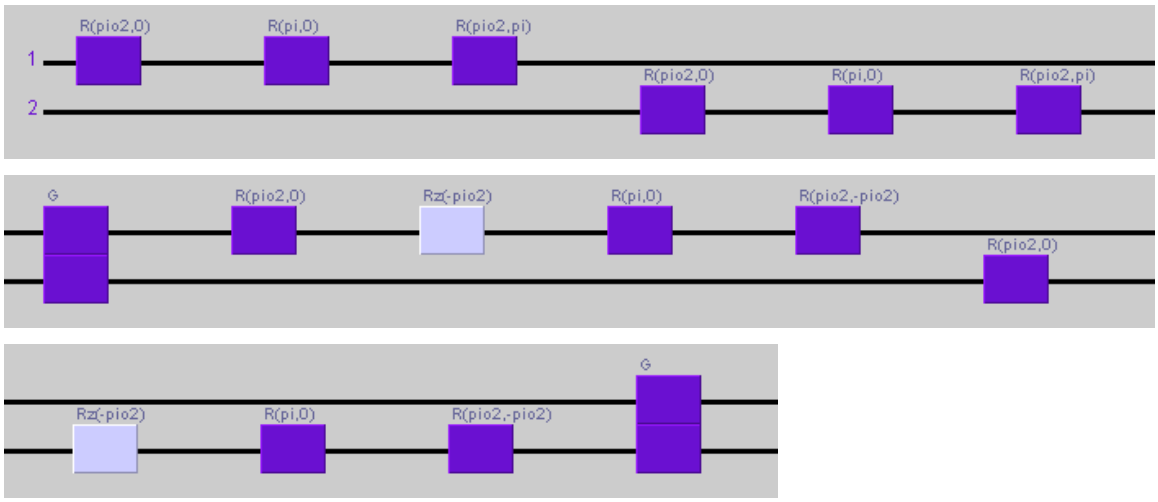

Figure 38: GUI View of a Grover Circuit

Table 11: Grover Simulator Results with No Noise

	Probabilities			
State	**|00> Target**	**|01> Target**	**|10> Target**	**|11> Target**
|00>	0.9999822	2.0313070E-15	2.0653559E-15	2.4729053E-15
|01>	5.0157198E-17	0.9999833	8.1217639E-17	2.4288804E-15
|10>	1.8642173E-16	1.5700924E-16	0.9999833	2.2963545E-15
|11>	1.8448619E-15	8.9420984E-17	1.0097672E-24	0.9999843

Table 11 shows the simulator's output for each case in the absence of gate noise and decoherence. Note that the probability that the target state is produced by the circuit is very close to unity, as expected. The deviations from the ideal results are due to numerical approximations and round-off errors.

Semi-classical QFT Algorithm

The purpose of the semi-classical QFT algorithm as experimentally implemented is to perform the quantum analogue of the discrete Fourier transform on a three-qubit state [13]. The general QFT on a computational basis state in an N-dimensional space is given by:

$$\left|k\right\rangle \longrightarrow \frac{1}{\sqrt{N}}\sum_{j=0}^{N-1} e^{i2\pi jk/N}\left|j\right\rangle . \tag{40}$$

The transformation rotates the basis state into a superposition of all the states in the space, with the given complex amplitudes.

The semi-classical implementation loses relative phase information for the output state but is experimentally much simpler to implement. The algorithm can intuitively be thought of as measuring the frequency components (or the periodicity) of the input state.

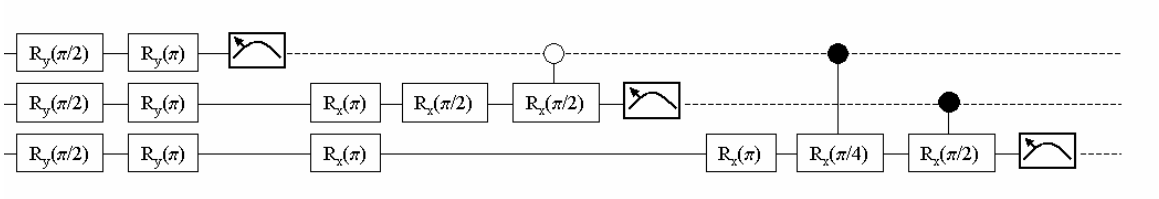

Figure 39: Experimental Implementation of the Semi-classical QFT Algorithm

Figure 39 shows the experimental implementation of the algorithm. The figure is a modified version of that published in [13]. In order to complete the testing, we had to work with our contact at NIST to translate their published algorithm into a standard quantum computing circuit diagram. The modifications have no effect on the operation of the algorithm, and the circuit design for the simulator is the same as in Figure 39.

The algorithm works by performing a number of single-qubit rotations to put the qubits into superposition states, and then performing a series of measurements and controlled rotations to collapse qubit states to classical information and selectively rotate the remaining qubits based on the measurement results.

In Figure 39 we used the short-hand notation

$$R_x(\theta) = R(\theta,0), \tag{41}$$

$$R_y(\theta) = R(\theta,\frac{\pi}{2}), \tag{42}$$

where $R(\theta,\phi)$ is the same as in the Deutsch-Jozsa circuit. Dashed lines denote classical information. Empty circles denote rotation if and only if the control qubit is |0>, and solid circles denote rotation if and only if the control qubit is |1>. The controlled rotations are performed using composite Hamiltonians expressed in terms of the carrier Hamiltonian:

$$H_{\text{empty}} = \begin{pmatrix} H^{\text{C}} & & 0 & 0 \\ & & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \tag{43}$$

$$H_{\text{solid}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & & \\ 0 & 0 & & H^{\text{C}} \end{pmatrix}. \tag{44}$$

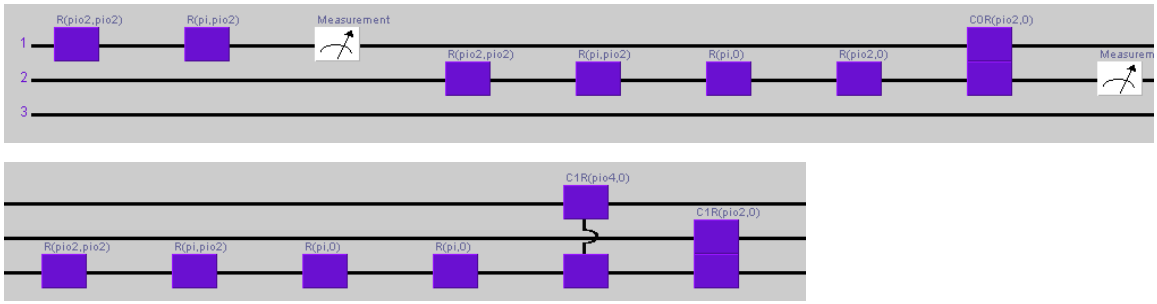

Figure 40: GUI View of the Semi-classical QFT Circuit

Again, the simulator cannot handle Hamiltonians with parameters in their matrix elements, so separate gates had to be constructed for each set of angle arguments. Figure 40 shows what the circuit looks like in the Quantum eXpress GUI. There is only one circuit for this algorithm, but there are five different input states, corresponding to states of period 1, 2, 3 (approximately), 4, and 8.

Table 12 shows the simulator's output for each input state in the absence of gate noise and decoherence. The input states are listed in the top row (in order of periodicity), and the output state probabilities are listed in the columns. Note that the circuit as performed in the experiment produces its output with the bits in reverse order. We have accounted for this in the table.

The periodicity of an input state can be determined by counting how many non-negligible output state probabilities there are for the input state. The non-negligible probabilities should be roughly equal and sum to unity. For example, the input state |011>+|111> has four non-negligible output probabilities, each approximately 0.25, indicating that this input state has a period of four. That

means that when stepping through the eight three-qubit basis states (listed in the first column of the table), every fourth state will be a component of the input state.

Table 12: Semi-classical QFT Simulator Results with No Noise

State	Equal Superposition (Period 1)	|001>+|011>+ |101>+|111> (Period 2)	|001>+|011>+ |100>+|110> (Period 3)	|011>+|111> (Period 4)	|111> (Period 8)
|000>	0.9999968	0.4999984	0.4999984	0.2499992	0.1249996
|001>	4.6116731E-18	2.7014556E-18	0.0366115	2.7014555E-18	0.1249996
|010>	1.9794813E-17	3.9420938E-17	4.9908754E-18	0.2499992	0.1249996
|011>	7.9123796E-19	2.7014555E-18	0.2133877	2.7014554E-18	0.1249996
|100>	5.1449158E-17	0.4999984	5.5992798E-17	0.2499992	0.1249996
|101>	7.9123804E-19	2.7014556E-18	0.2133877	2.7014555E-18	0.1249996
|110>	1.9794814E-17	3.9420938E-17	4.9908760E-18	0.2499992	0.1249996
|111>	4.6116729E-18	2.7014554E-18	0.0366115	2.7014554E-18	0.1249996

Note that there is no true period-three state in an eight-dimensional space, so the results for that input state are approximate. That is why it has three large output probabilities and two small ones. The other deviations from perfection are due to numerical approximations and round-off errors.

4.4.3 Results Analysis

Comparison with Ideal Results

We compared the simulation results in the absence of gate noise and decoherence with the ideal results for all cases of each algorithm. By "ideal results," we mean the theoretical results of the algorithm, which is what would be obtained in the absence of noise, decoherence, experimental error, etc. For example, an ideal implementation of the Grover algorithm would find the marked state (i.e. produce it as the output of the algorithm) 100% of the time.

Table 13: Simulation Maximum Absolute Probability Error with No Noise (%)

Case	Deutsch-Jozsa	Grover	Semi-classical QFT
1	0.000006	0.002	0.0003
2	0.0003	0.002	0.0002
3	0.00007	0.002	0.0002
4	0.0002	0.002	0.00008
5			0.00004

Based on the approximations that Quantum eXpress makes, we expected that the simulator results should match the ideal results with less than 0.01% absolute probability error. Table 13 shows the maximum absolute probability error for all states for each case of each algorithm. All the errors are well below the expected threshold. The maximum of all of the errors is 0.002%, for the Grover algorithm. This indicates that in the absence of gate noise and decoherence, Quantum eXpress accurately models the ion trap quantum computer algorithms that we selected.

Comparison with Experimental Results

To prepare for comparison with experimental results, we simulated each case of each algorithm with each gate noise level varying from 0% to 20% in increments of 1% simultaneously. The noise percentage defines the standard deviation of a Gaussian distribution of gate application times in terms of the target time. We ran each simulation 450 times to get a statistical distribution of results.

To control the simulations, we wrote MATLAB code to perform the following tasks:

- update the XML files for the desired noise level
- run the simulator
- extract the probabilities of the output states
- compare the results to the ideal cases and the experiments

We made two comparisons with the experimental results. First we compared the fidelities of the experiments with the fidelities of the simulations. For this comparison, we used gate noise levels of 2% for the Deutsch-Jozsa and Grover algorithms, and 1% for the semi-classical QFT. These numbers are based on estimates we received from the experimentalists. By "fidelity," for the Deutsch-Jozsa and Grover algorithms we mean the probability that the algorithm produced the correct result (e.g., did the Deutsch-Jozsa algorithm correctly determine whether the function was constant or balanced). For the semi-classical QFT, we used the squared statistical overlap (SSO) metric that the experimentalists used [13]:

$$\gamma = \left(\sum_{j=0}^{7} \sqrt{m_j e_j} \right)^2, \tag{45}$$

Table 14: Experimental and Simulation Fidelities (%)

Case	Deutsch-Jozsa		Grover		Semi-classical QFT	
	Exp	QX	Exp	QX	Exp	QX
1	98.1(6)	99.947(3)	64(2)	98.21(7)	87(1)	99.828(5)
2	91.3(6)	99.75(1)	62(2)	98.20(7)	88(1)	99.867(4)
3	97.5(4)	99.877(5)	49(2)	98.18(7)	88(1)	99.867(4)
4	97.5(2)	99.881(4)	65(2)	98.35(6)	96(1)	99.919(3)
5					99(1)	99.940(3)

where m_j and e_j are the measured and expected probabilities of basis state j, respectively. The SSO is a measure of the similarity of two states in the Hilbert space and varies from 0 to 1.

Table 14 shows the experimental and simulation fidelities. The numbers in parentheses are the one-standard-deviation uncertainties in the final digits. The simulations generally have higher fidelities than the experiments. This was

expected because we did not model decoherence or systematic errors in state preparation or measurement. The experimental fidelities are lowest for the Grover algorithm (especially the third case), which apparently is the most difficult algorithm to implement. The fidelity of the semi-classical QFT experiment for the period-eight state is particularly high, probably because the initialization of this state is simple.

For the second comparison with experiment, we determined the minimum gate noises that were necessary for the experimental and simulation fidelities to be equivalent (see Table 15). We considered two fidelities to be equivalent if their one-standard-deviation ranges overlapped. We determined the required noise independently for each case of each algorithm. These increased gate noises are our attempt to model the additional experimental error sources in the absence of quantitative information on decoherence.

Table 15: Minimum Gate Noise for Comparable Experimental and Simulation Fidelities

Case	Deutsch-Jozsa			Grover			Semi-classical QFT		
	Exp	QX	Noise	Exp	QX	Noise	Exp	QX	Noise
1	98.1(6)	98.55(9)	11	64(2)	64(1)	10	87(1)	86.8(4)	9
2	91.3(6)	91.7(3)	12	62(2)	63(1)	11	88(1)	88.6(4)	10
3	97.5(4)	97.55(9)	9	49(2)	51(1)	13	88(1)	88.8(4)	10
4	97.5(2)	97.4(1)	9	65(2)	65(1)	11	96(1)	96.3(1)	7
5							99(1)	99.9997	0

The table shows the experimental fidelities, the comparable simulation fidelities, and the gate noise levels that produced the simulation fidelities. All the entries other than the case indices are percentages. The results indicate that we generally require about 10% noise in the gate application times for the experimental and simulation results to be comparable. The poor fidelity for the third case of the Grover algorithm requires a higher noise level (13%), and the excellent fidelity for the period-eight state of the semi-classical QFT algorithm does not require any noise.

4.5 Field Programmable Gate Array-Accelerated Simulation

As described previously, Quantum eXpress uses the density matrix representation of the quantum state, requiring a $2^N x 2^N$ matrix to represent an N-qubit system. Since the size of the density matrix grows exponentially in the number of qubits, the size of the matrix multiplications needed to simulate a quantum gate application also grows exponentially. This creates a tremendous computational burden, making it important to explore alternative architectures for the quantum simulator. Field Programmable Gate Arrays (FPGAs) are one such alternative architecture [15].

FPGAs are semiconductor-based devices that contain user programmable logic components. These components provide basic logic functions such as AND, OR, XOR, and NOT functions. Starting with a blank piece of silicon, large and

complex systems can be constructed from these components. By carefully crafting the memory architecture, an FPGA design can avoid the Von-Neumann bottleneck that plagues general purpose processors (GPPs). The Von-Neumann bottleneck refers to the limited bandwidth between the CPU and main memory; only one memory request can be completed at a time. Since FPGAs can avoid the sequential memory access limitation, they are very attractive for executing parallelized algorithms. The major drawback of FPGAs is the time and effort required to design and implement a highly optimized algorithm.

Prior Art

As the interest surrounding quantum computing has increased, researchers have been developing various simulators, a few of which were based on FPGAs. The scope of this research has been limited however, as many of the simulators are applicable to only a narrow range of quantum algorithms.

Negovetic et al. of the Portland Quantum Logic Group at Portland State University developed an FPGA-based quantum emulator [16]. Their emulator was implemented using Verilog and can model quantum algorithms with one or two qubits. Their emulator was evaluated on an algorithm that consisted of two inverters and a Hadamard gate.

Fujishima, along with other researchers at the School of Frontier Sciences at the University of Tokyo, developed their own emulator [17]. Theirs works as a quantum index processor (QIP), which means that the quantum states are not stored, only the indices of the "1"'s in the quantum state. This limits the operation of the emulator, as it must use pre-defined lookup tables to simulate a quantum gate.

Finally, a group from McGill University also developed a quantum algorithm emulator in an FPGA [18]. In their approach, they model quantum algorithms as "quantum circuits." These quantum circuits are generated using scripts the group created. These scripts output VHDL, which can then be compiled and executed on an FPGA. They support Walsh-Hadamard, Phase Shift, X, CNot, and Z gates.

4.5.1 FPGA Hardware & Tools

GE Global Research has had prior experience using FPGA's to optimize a diverse set of applications. From this experience, GEGR has built up an array of FPGA hardware and design tools, which were leveraged to design and simulate an FPGA-based implementation of the Quantum eXpress engine.

FPGA Hardware

The objective of this work was to develop and simulate an FPGA design that would provide an accurate understanding of real-world performance. To achieve this objective the design was targeted to a particular FPGA device, setting constraints such as the amount of memory addressable, the number of

multipliers that could be instantiated, and the clock speed at which the design would operate. Quantum eXpress is very intense in terms of complex floating point arithmetic. Floating point multiplications in particular use considerable FPGA design space, making it necessary to choose a device that was relatively large in terms of the number of resources available.

The FPGA device chosen was a Xilinx xc2vp100 on a BenBLUE-III module [19]. The xc2vp100 FPGA is in the Xilinx Virtex-II Pro family of parts. It contains 99,216 logic cells along with 7,992 Kbits of Block RAM. This is the second largest FPGA in the Xilinx Virtex-II Pro family. Paired with the BenBLUE-III, the module contains 64 MBytes of Zero Bus Turnaround (ZBT) SRAM in 8 independent banks with a data rate of 166 MHz. This whole module resides on a Nallatech Bennuey-4E PCI motherboard [20].

Of particular importance is the ZBT SRAM. This memory is where the density matrix and Hamiltonian matrices are stored. ZBT SRAM was used for two reasons. First, these matrices are too large to fit into the FPGA's BRAM with enough space left over to instantiate the design logic necessary for the quantum simulator. Second, the ZBT (Zero Bus Turnaround) SRAM is fast–there is no bus latency associated with the memory, and therefore there are no overhead cycles when accessing the SRAM.

The Virtex-II Pro family, including the xc2vp100 chosen for this project, is based on a CMOS SRAM fabric. Unlike other FPGAs based on technologies such as anti-fuse, SRAM-based devices can be repeatedly reprogrammed. Reconfigurability is advantageous for a number of reasons. First, it is particularly useful during the debugging phase. If a design that was programmed onto an FPGA contained errors, the FPGA can simply be re-written rather than discarded. This is also useful since the FPGA can be upgraded in the field. Another advantage comes from the fact that multiple FPGA designs can be created to perform specific tasks. When a specific task is required, its associated design can be loaded onto the FPGA.

FPGA Design Tools

Along with access to hardware, GEGR also has licenses available to several software packages that aid in the development of FPGA designs. This includes Nallatech's DIMETalk 3 software [21]. DIMETalk is a GUI-based FPGA design environment. It allows users to build communication networks between algorithm nodes, memory, and interfaces to other FPGAs and other computing systems.

Also used was Nallatech's DIME-C, which is a C-to-VHDL functional translator. This was available through an early access program as the software is not yet freely available. DIME-C creates VHDL components that plug into the DIMETalk framework.

Initially, CVER, an open-source Verilog simulator, was used for simulating the FPGA accelerator [22]. However, Nallatech's tools generate VHDL code, not Verilog. Fortunately an evaluation license of Active-HDL 7.1 was obtained from Aldec, replacing CVER and allowing us to simulate directly from the VHDL code [23].

4.5.2 FPGA Design

The FPGA's greatest strength comes from its ability to take advantage of fine-grained parallelism in designs. This parallelism is necessary to get any sort of speed-up in a design ported to an FPGA. FPGAs run at much lower clock rates than their general purpose processing counterparts. The maximum clock rate for most designs using the Xilinx Virtex-II Pro family of devices is less than 200 MHz, while the latest offerings for general purpose processors (GPPs) are all over 2 GHz. This means the same algorithm implemented without parallelism will execute significantly faster on a GPP than on an FPGA.

FPGA Architecture for Quantum eXpress

In determining what aspect of the simulator to translate to the FPGA, it was determined that the gate application portion of the design (covered in Section 4.1.2) was most suited. This is because the complex matrix multiplications in that step can be highly parallelized. To achieve a high degree of parallelism the matrices must be stored on fast memory accessible to the FPGA. Thus, the memory and memory controller were placed on the FPGA, as well, as shown in Figure 41. This reduced the data transfers from the host PC and the FPGA.

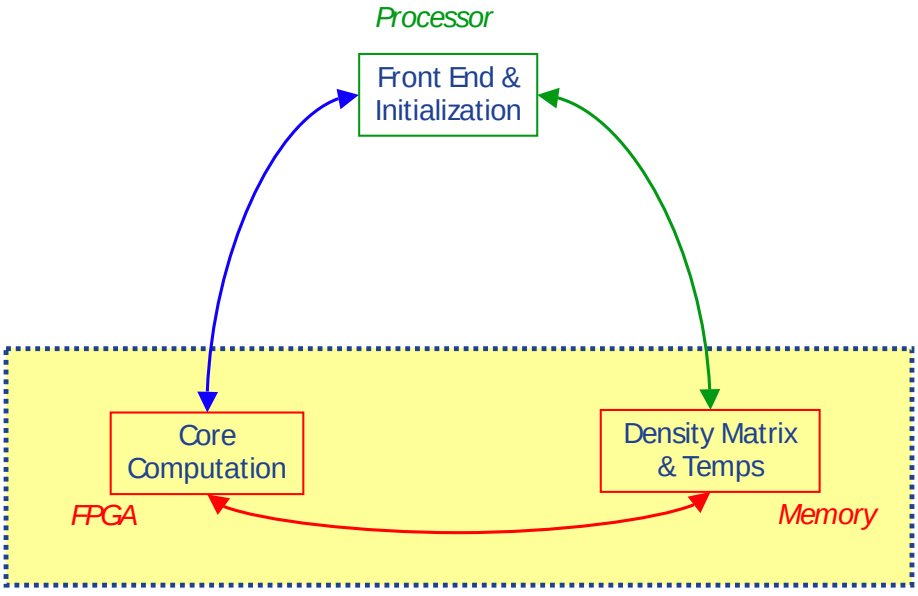

Figure 41: FPGA & GPP Simulator Architecture

The general-purpose QX implementation starts with three inputs and consists of four steps. The inputs are the density matrix, the Hamiltonian matrix, and the

gate indices to which the quantum gate is applied. The indices for the gate are not necessarily in sequential order and may affect indices distributed throughout the density matrix. This can be a serious detriment to performance on a general-purpose processor. As an optimization, Quantum eXpress reorders the density matrix so that the bits that will be operated on share a continuous memory space.

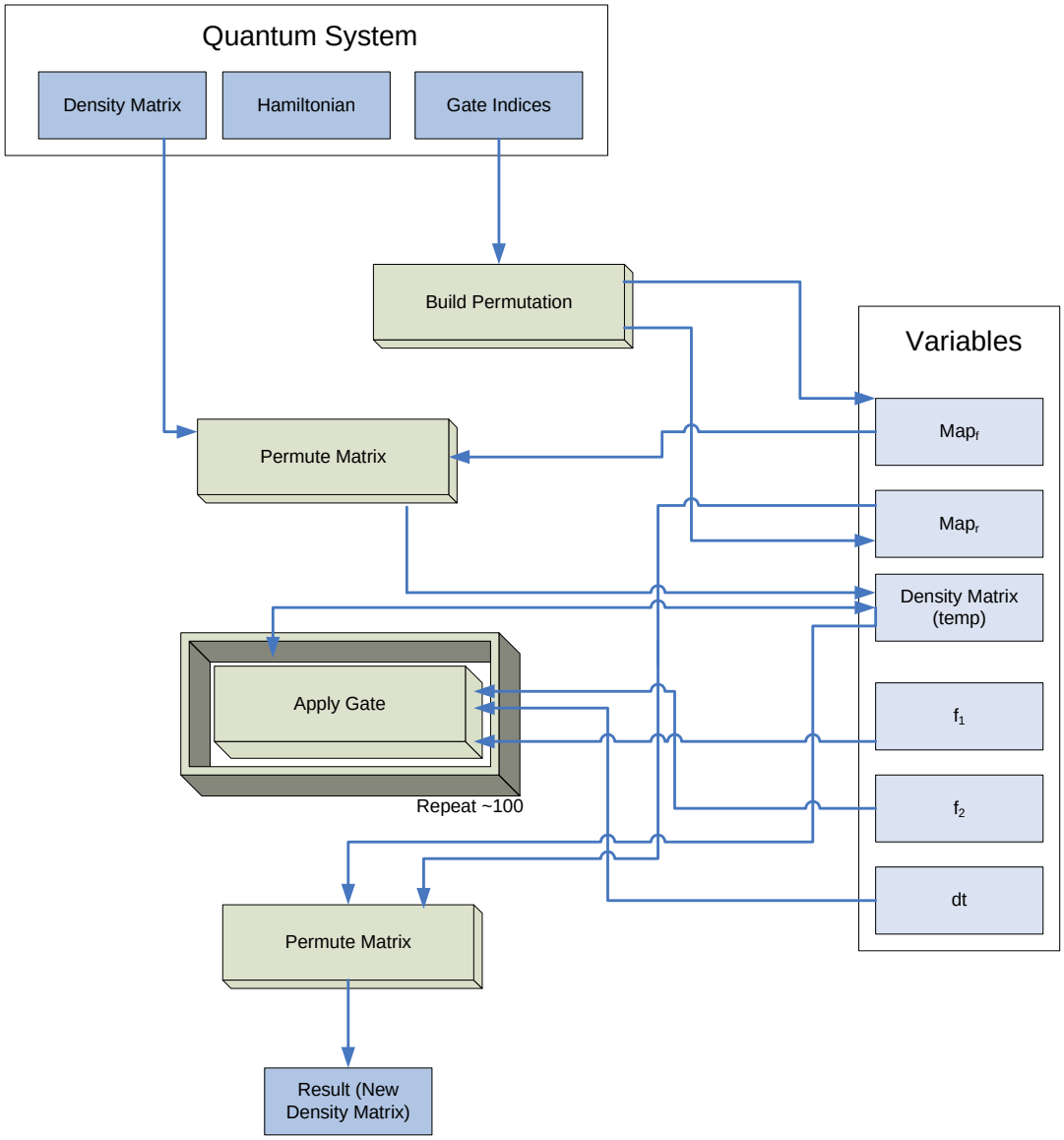

Figure 42: Architecture of Original GPP Implementation

After the density matrix is in standard form, the gate application computation is performed, as shown in Figure 42. This is the most computationally intense part of the simulation, consisting of a series of complex matrix multiplications. The gate application routine is performed on the density matrix 100 times. Finally, the density matrix is permuted again to bring it back to its original form and the computation is complete.

An advantage of the FPGA-based implementation is that there is no need to re-arrange the density matrix before performing the gate application. An FPGA has no cache. As such, there is no penalty for a cache miss like on a general-purpose processor. On an FPGA the time to read and write to a memory location is constant, independent of the address that is being referenced. These factors have the effect of greatly reducing the complexity of the FPGA architecture.

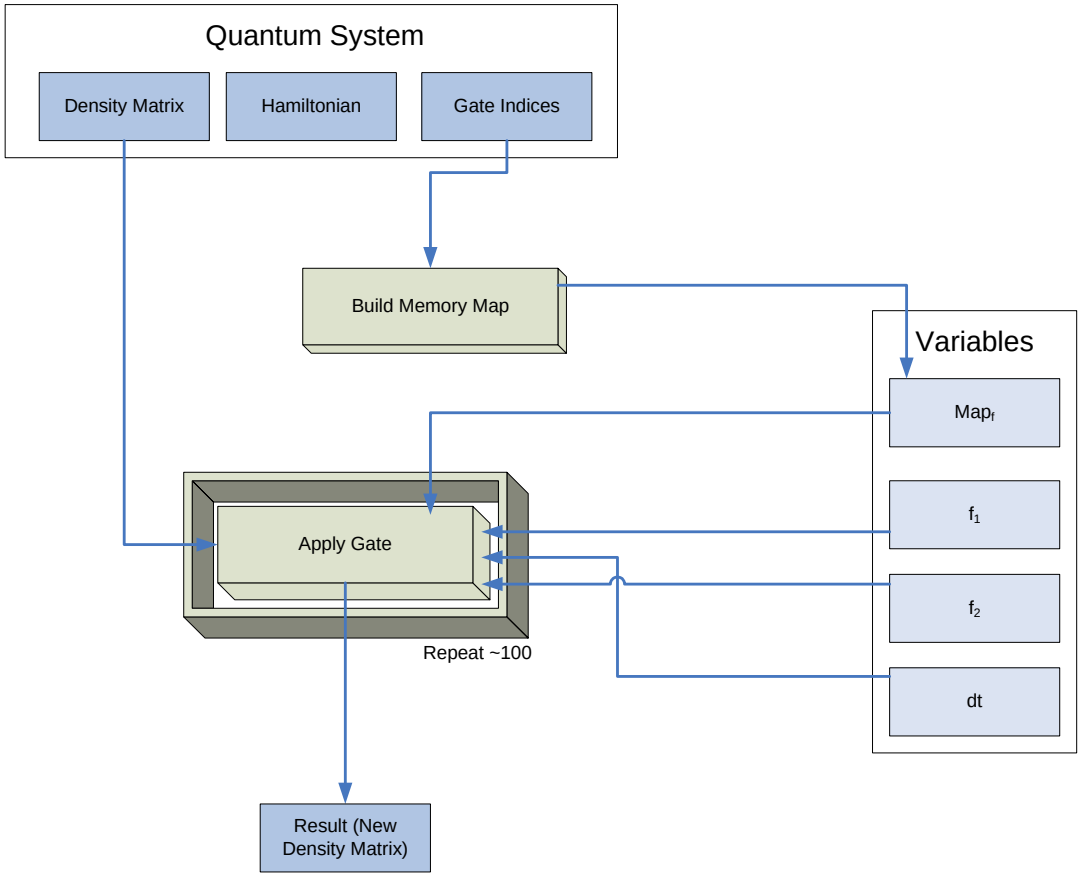

Figure 43: Architecture of FPGA Implementation

For the FPGA-based implementation, a memory map is still created as shown in Figure 43. The mapping is used to make the density matrix appear sequential in terms of the gate application, but the physical location of the bits in memory are not changed. This greatly simplifies the gate application algorithm. And since the density matrix does not have to be permuted, a copy does not have to be created, saving both time and memory.

Another major architecture change lies in the gate application computation. Originally, this consisted of a single operating thread that computed each element in the resulting density matrix serially. Taking advantage of the fine-grained parallelism of the FPGA, multiple operations can be computed at once. This is particularly useful for complex matrix multiplications. For example, the

individual element-wise multiplications of the real and imaginary parts can be computed simultaneously. Results of the intermediate steps are then summed to get the final complex multiplication result. Further, for a gate matrix of size $2^g x 2^g$, 2^g element-wise multiplications can be computed at once. This offers significant savings; particularly as the gate size g increases. In the original complex multiplication implementation the simple element-wise multiplication needed to be iterated over 2^{2g} times. After the optimization, these 2^{2g} iterations were replaced with 2^g complex multiplications, dramatically reducing the total number of operations necessary to complete one cycle of the gate application. Figure 44 illustrates the reduction in the number of iterations necessary to apply a gate operating on two qubits. Figure 45 shows the iteration reduction for a gate size of three qubits. For both cases, as the number of qubits in the system increases, the effect of the optimization becomes more and more pronounced.

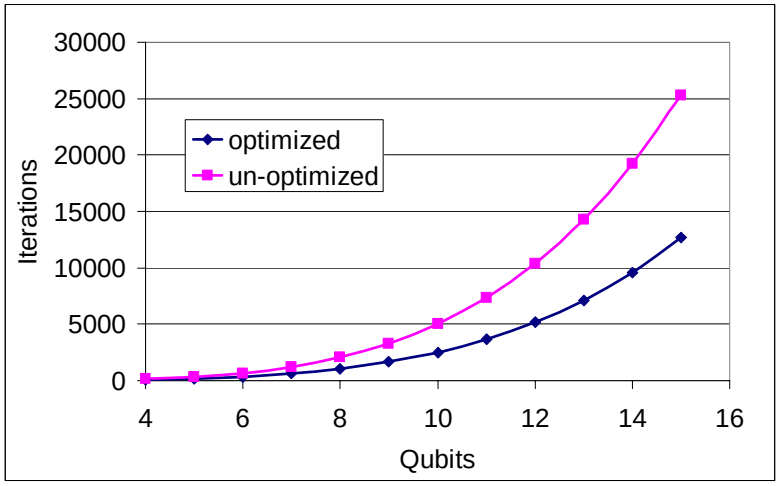

Figure 44: Iteration Reduction in Evaluating Master Equation for Gate Size g = 2

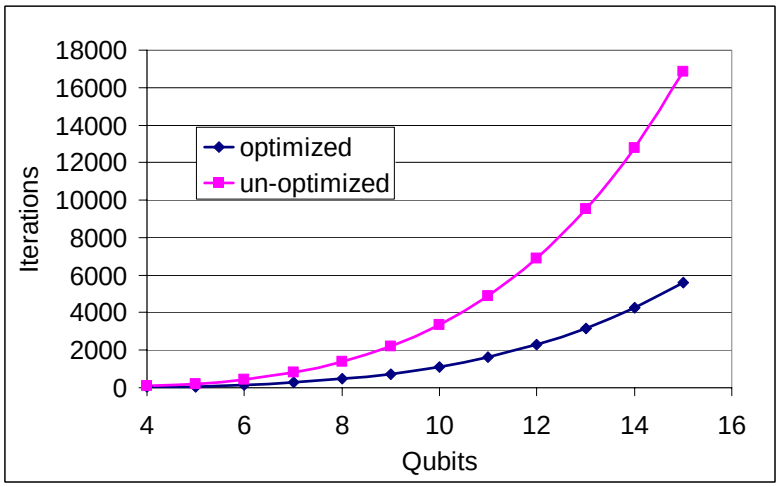

Figure 45: Iteration Reduction in Evaluating Master Equation for Gate Size g = 3

The fine-grained parallelism affects the pipeline of the FPGA. As expected, the number of operations that occur in parallel dramatically increases, increasing the FPGA's resource utilization. Pre- and post-optimization pipeline diagrams generated by Nallatech's DIMEC tools are shown in Figure 46 and Figure 47, respectively, showing the pipeline length in the horizontal axis and the parallel operations in the vertical axis.

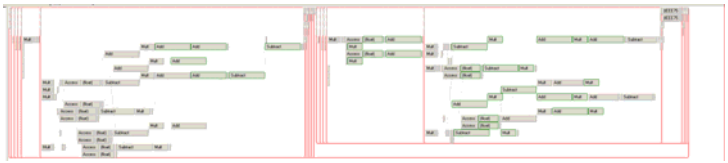

Figure 46: Un-Optimized Pipeline

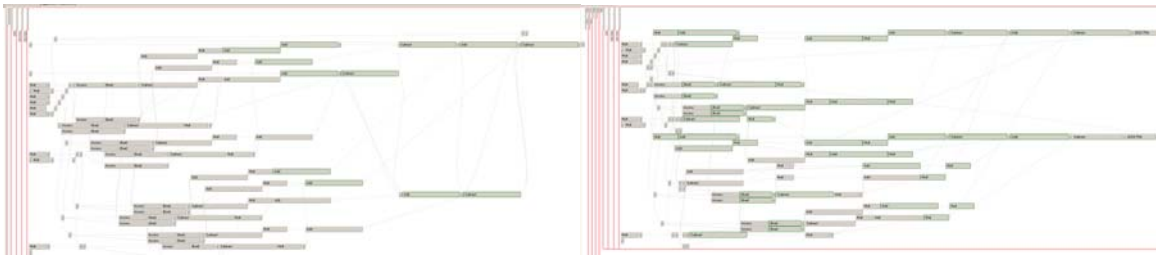

Figure 47: Optimized Pipeline

The increase in the pipeline width due to the parallelism from loop unrolling is quite noticeable. For the initial QX implementation, iterating over the same code many times performed much of the computation. After the FPGA optimizations many more operations occur in parallel. Towards the end of each of the loops in the pipeline diagram the number of parallel operations converge until there is only one operation occurring—the results from the previous operations are combined to produce a single element in the density matrix.

The design of the FPGA-accelerated quantum simulator was done using DIMETalk and DIMEC. DIMEC was used to develop the computationally intense gate application portion of the design. DIMEC claims to be an ANSI C-to-VHDL functional translator, however, it currently suffers from a number of limitations. DIMEC is unable to parse ANSI C pointers, multi-dimensional arrays, structures, unions, enumerated types, switch statements, and do-while loops. It was rather awkward working around its gaps in the ANSI C specification, especially when trying to port previously developed source code. Even with these restrictions DIMEC is very powerful for rapidly implementing FPGA-accelerated algorithms, however.

Using C as a code base was particularly beneficial for several reasons. Standard tools can be used for the initial design, such as a familiar C development

environment. Debugging the initial implementation was easier because the availability of a standard C compiler and debugging tools.

After the algorithm was validated, it was modified to fit conventions of DIMEC. These conventions include specifying the inputs and outputs to the algorithm along with the storage format of these parameters. The parameters consist of the density matrix, gate Hamiltonian, time for each iteration of the gate application, and the number of times to iterate. The density matrix and the gate Hamiltonian are stored on the on-board, off-chip ZBT SRAM, as mentioned previously. It should be noted that the data type used for storing data in SRAM is slightly different than that used for performing the computations on the FPGA. These differences are described in Appendix A – FPGA Floating Point Formats.

After creating the DIMEC gate application module it was placed into the DIMETalk environment. The DIMETalk design environment allows the user to create a system communication diagram to specify the high-level design. This is done by first specifying the communication host, in this case a PCI host interface. From there, the system components are laid out. Four ZBT SRAM memory nodes were created, one for each of the real and imaginary components of the density matrix and Hamiltonian. A memory map is used by the DIMEC component for access to BRAMs. Routers are instantiated to handle the communication between all of the DIMETalk nodes. Finally, the design is "wrapped" with a description of the hardware onto which it will be programmed, as shown in Figure 48.

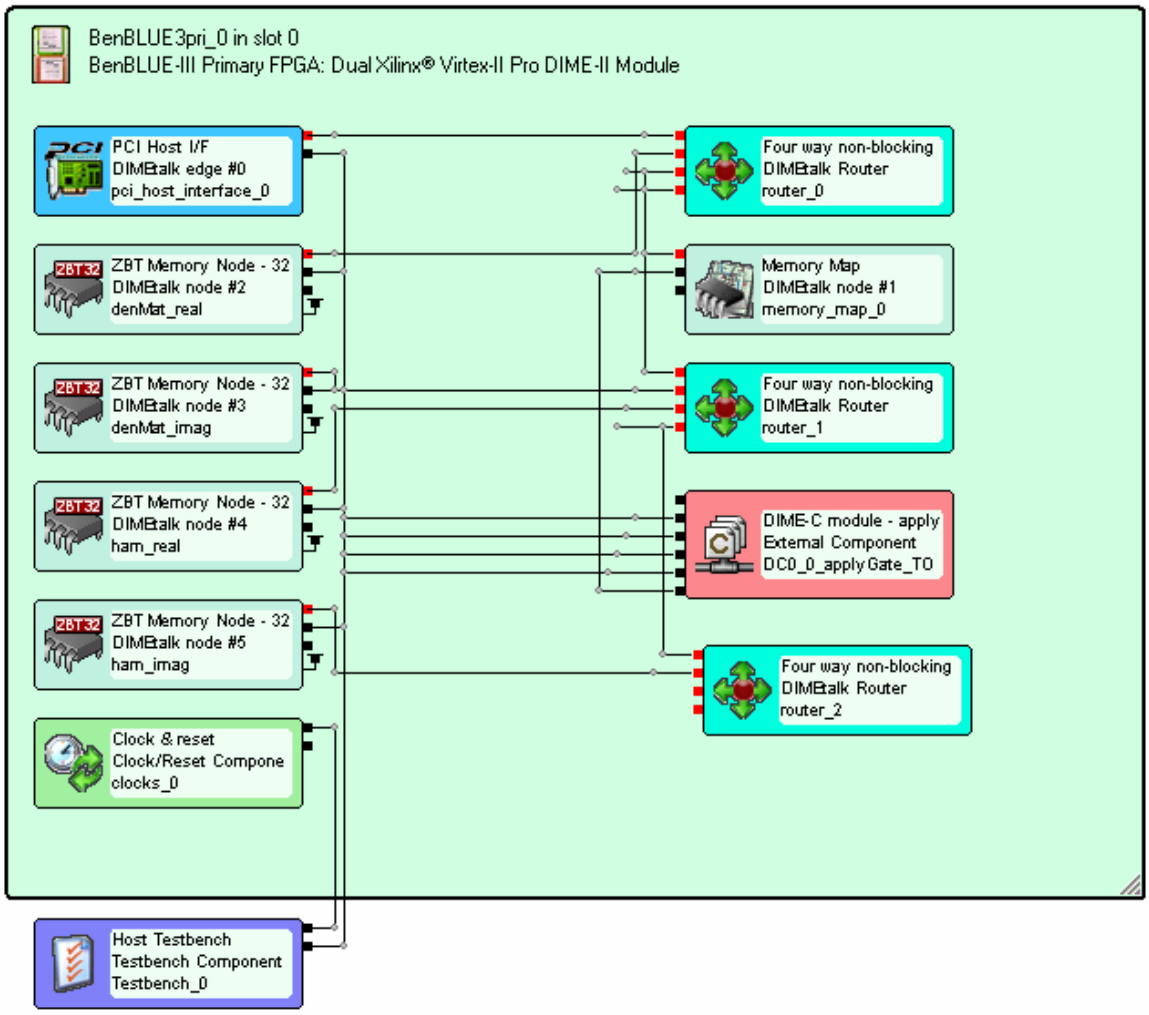

Figure 48: DIMETalk Diagram for the FPGA Accelerator

After creating this diagram, DIMETalk generates a VHDL description of the entire system. These components can then be synthesized, placed, routed, and put onto a hardware device. None of the I/O associated with the design has been constrained to the hardware, however. For it to actually operate on a hardware device, the I/O must be assigned to specific pins on the FPGA.

4.5.3 Design Limitations

The first major limitation of the design is that it can only simulate quantum systems that are 6 qubits or smaller. This limitation comes from the fact that the device selected, the BenBlue III, contains 64 MBytes of ZBT SRAM in 8 independent banks. At 32 bits per element in the density matrix, the largest system that could be stored in a single bank of SRAM was 6 qubits. This is with the density matrix split so that its real and imaginary components occupy separate memory banks. If a more complex memory controller was created, a larger density matrix could be stored that spans different banks in the SRAM.

Another limitation of this design is that the number of qubits in the system is fixed at design time and cannot be changed dynamically at run time with no loss of resources. In order to simulate quantum systems with different sizes of qubits and different gate sizes there are two options.

The first option involves creating an overly large design with the maximum number of qubits and gate sizes. This design would be created and loaded onto the FPGA for each gate simulation. A parameter would identify how many of the total qubits are actually being utilized. The design would then ignore the unused qubits. However, by loading the FPGA with an overly large design, the design itself is not optimal and there is overhead from performing needless computations. Also, since resources on the FPGA are being allocated for no reason, there is an additional impact from unnecessary power consumption. This option was chosen for implementation in this research effort.

The second option would involve pre-designing, creating, and compiling bit-files for every possible system size that could be simulated. These bit-files could be stored in a repository and programmed on the FPGA just before simulation. This scheme provides the most hardware efficiency at the cost of a significant amount of up-front work to create all of the bit-files.

4.5.4 Testing Procedure

The FPGA-based design was simulated using Aldec's Active-HDL 7.1, an easy-to-use simulation environment for VHDL. Two of the three test cases from Section 4.1.1 could fit within the FPGA simulator design: the 3 qubit and 5 qubit ones. The Active HDL simulator was paused after each gate application routine had completed, to record its execution time. This time was benchmarked against the execution time of the gate application portion of the single processor implementation of Quantum eXpress. Quantum eXpress was run on an HP wx9300 workstation based on an AMD Opteron 280 dual processor dual core system clocked at 2.4 GHz. The system had 4GB of RAM running Windows XP Professional x64 Edition and Sun's Java 1.5.0-b64 JVM, a high-end workstation at the time of this writing.

The time used to compare the GPP and FPGA simulations was the sum of the gate application times. Additional overhead for the algorithm configuration, such as data load time, was not considered. The speedup of the FPGA vs. the single processor implementation was calculated using the equation:

$$\text{Speed - up} = \frac{\sum \text{QX gate application times}}{\sum \text{FPGA simulation gate application times}}. \tag{46}$$

4.5.5 Results

Figure 49 shows the speed-ups achieved for the two algorithms. It was found that the FPGA simulator took roughly 10ms to apply each CNOT gate in the 3-qubit test case. For the total algorithm, the FPGA speedup over the single processor implementation of Quantum eXpress was 12.4x. For the 5-qubit test case, the speedup of the FPGA implementation was found to be 26.9x over the single processor implementation.

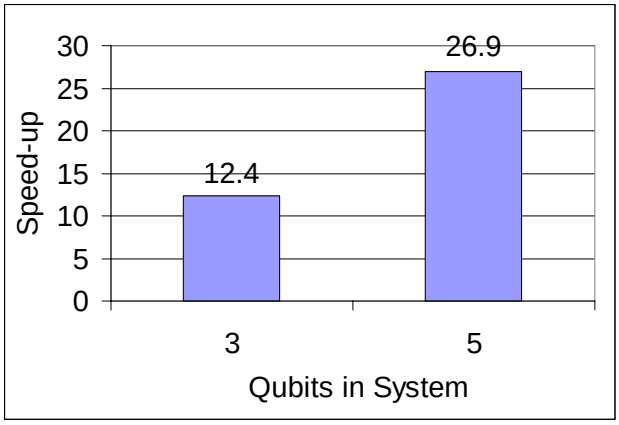

Figure 49: Speed-up of FPGA vs. Single Processor GPP Implementation

For the 3-qubit test case all of the gates in the algorithm are identical. The speedup for each individual gate was 12.4x and thus, aggregating this speedup across the algorithm yields the same speedup.

The 5-qubit test case is composed of two types of gates. The first set is three CCNOT gates that have a gate width of 3. The second set is 3 CNOT gates that operate on 2 of the 5 qubits in the system. The average speedup for the 3-qubit CCNOT gates was 35.5x and the speedup for the 2-qubit CNOT gates was 18.2x. As expected, as the number of qubits in the system increased, so did the speedup. Additionally, as the gate width increased, the speedup also increased. The FPGA accelerator was designed to take advantage of the fine-grained parallelism available during the gate application. As the gate width increases, there is more opportunity for parallelism and thus we get a bigger speedup. The FPGA implementation appears to scale very well as the size of the quantum system increases.

Single vs. Double Precision

All of the floating point cores used in the FPGA implementation are single precision. In Quantum eXpress, all of the mathematical operations are performed using double precision floating point. In order to assess the impact of reducing the precision, we calculated the root mean-squared error (RMSE) across all of the final state measurements. This came to $3.08*10^{-8}$ for the 3-qubit test case, the errors for which are shown in Table 16.

Table 16: Precision Comparison for 3 Qubit Test Case

Quantum State	GPP double	FPGA single	Error
|000>	0.5	0.5	0
|001>	0	0	0
|010>	0	0	0
|011>	0	0	0
|100>	9.64E-05	9.64E-05	-1.09539E-09
|101>	0.499661	0.499661	4.7733E-08
|110>	1.96E-04	1.96E-04	1.21768E-08
|111>	4.67E-05	4.68E-05	-7.20057E-08
	sum = 1	sum = 1	**RMSE = 3.08479E-08**

For the second test case, the root mean-squared error came to $3.45*10^{-6}$, with the individual state errors shown in Table 17. The RMSE for this case was several orders of magnitude greater than in the previous test case. This is because after each gate in an algorithm, the loss in precision is being aggregated and the errors become more pronounced. At this stage, the error is still insignificant but for larger algorithms, the error could accumulate to unacceptable levels. At what size algorithm the errors become unacceptable requires further investigation.

Table 17: Precision Comparison for 5 Qubit Test Case

Quantum State	GPP double	FPGA single	Error
|00000>	0	0	0
|11100>	4.64E-07	4.61E-07	3.15396E-09
|11101>	6.83E-04	6.77E-04	5.24537E-06
|11110>	6.79E-04	6.79E-04	-4.32934E-07
|11111>	0.998638	0.998644	-5.63888E-06
	sum = 1	sum = 1.000001	**RMSE = 3.44959E-06**

5.0 CONCLUSIONS

We were able to significantly improve the performance of Quantum eXpress by focusing our attention on optimizing the key operations performed by the simulator: large matrix multiplications. By taking advantage of the structure of the matrices used, we were able to reduce both the number of calculations performed and the memory requirements. A third benefit of the optimizations was the resulting operations naturally lent themselves to a parallelized evaluation, so we were able to distribute the simulator computation across a high performance computing cluster. The optimizations and subsequent distribution across a cluster significantly improved the performance of the simulator. Now, quantum algorithm researchers can use Quantum eXpress to exercise and quantify the performance of quantum algorithms with both decoherence and noise in an interactive environment.

We were also able to enhance the simulator to calculate a step-by-step comparison of the ideal quantum algorithm execution to the actual (decoherence

and/or noise-affected) simulation. These differences can be visualized as heatmaps in the enhanced Quantum eXpress graphical user interface, allowing researchers to better study the effects of errors and error propagation in a quantum system.

We found that Quantum eXpress is capable of accurately modeling ion trap quantum computers. Due to experimental limitations, we were unable to test all of the simulator's capabilities (decoherence, qutrits, algorithms with four or more qubits, etc.), but it performed well compared to the most advanced publicly available experimental results. In particular, compared to the ideal algorithms, and in the absence of noise, the simulator's absolute probability errors were within 0.002%. We also found that the simulator is more accurate than the experiments when decoherence and systematic errors in state preparation and measurement are not modeled. We found that the simulation and experimental fidelities are comparable when gate noises of about 10% of the target application times are used.

Finally, we were able to create an FPGA-based quantum accelerator that took advantage of the fine-grained parallelism that is present during quantum gate applications. This yielded a significant speed-up in a 3-qubit algorithm and an even more substantial speed-up for a 5-qubit algorithm. Additionally, the effects of simulating the quantum algorithms using single precision floating point was compared to simulating with double precision. It was determined that the precision reduction in the data type had a negligible effect on the results of the two test cases.

5.1 Proposed Future Work

The major problem faced by Quantum eXpress or any density matrix-based simulation is the size of the representation of the quantum state, which is 2^{2N}. This manifests itself in three ways. The first manifestation is in the amount of memory required to store the state and the amount of temporary memory required to compute the state update. Second, it manifests itself in the time required to update the state. The third manifestation is the volume of information generated during a simulation, which could be useful for debugging and understanding error correction strategies. In this work we addressed the first two issues.

The remaining execution time issue to be addressed is the efficient simulation of amplitude decoherence. This represents a challenge because it does not maintain the locality demonstrated above for phase decoherence. There is hope, however, because the resulting computation does exhibit a very well behaved non-local reference pattern.

The next major challenge is posed by the development of tools for storing and examining the results of a simulation. The volume of data generated during the running of even a simple quantum algorithm is astronomical. The user will need

new tools to help in the identification of areas of interest and agile means for manipulating the data to bring those areas into view.

The focus of the FPGA accelerator effort was on taking advantage of the fine-grained parallelism inherent in the quantum gate application algorithm. There is still a lot of room for improvement in terms of performance, however. Most notably would be to create multiple execution kernels on the FPGA.

In the current implementation, there is only one active execution kernel on the FPGA. However, many more could be easily added. The algorithm can be parallelized in the same way the distributed simulation was configured, by distributing portions of the density matrix and Hamiltonian over several nodes. This scheme would still hold on an FPGA-based architecture. Each node could be a kernel running on an FPGA. Additionally, this scheme could be used to span multiple FPGAs. As long as each FPGA receives their portion of the density matrix there is no need for the kernels to all run on the same device.

Using the distributed scheme across multiple FPGAs has the added benefit of being able to handle quantum systems that have a larger number of qubits. The whole density matrix does not have to reside on a single FPGA, alleviating the memory constraints that limited the current implementation to simulations with 6 qubits. A design incorporating multiple FPGAs could allow us to simulate more qubits than possible using a cluster of conventional computing nodes.

Finally, the FPGA design was only simulated—a natural next step would be to test this design on a physical field programmable gate array device.

6.0 ACKNOWLEDGEMENTS

The authors would like to thank Dr. Lenore Mullin of the State University of New York, Albany for many discussions of the Psi calculus and how it could be used to view the fundamental structures in the simulation [24]. The Psi calculus supports the explicit manipulation of dimensions of an array in a way that simplifies much of what was presented in Section 4.1.2.

The authors would also like to thank Steven Drager and Captain Earl Bednar of the Air Force Research Laboratory for their guidance in shaping this research effort, and for time spent configuring and testing Quantum eXpress on the AFRL cluster.

7.0 REFERENCES

1. P.W. Shor, Algorithms for Quantum Computation: Discrete Logarithms and Factoring, *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, p124-134, 1994.

2. K.S. Aggour, M.J. Simon, R. Guhde, and M.K. Simmons, Simulating Quantum Computing: Quantum eXpress. *Proceedings of the Winter Simulation Conference*, v1, p932-940, New Orleans, LA, USA, 2003.

3. D.J. Griffiths, *Introduction to Quantum Mechanics*. New Jersey: Prentice Hall, 1995.

4. W.H. Louisell, *Quantum Statistical Properties of Radiation*. New York: John Wiley and Sons, 1973.

5. R. Schnathorst, *Numerical simulation of master equations*, internal communication, May 1, 2003.

6. Z. Meglicki, Introduction to Quantum Computing, http://beige.ucs.indiana.edu/M743/, last accessed October 16, 2006.

7. R. Schnathorst, *Decoherence: Examples and test cases*, internal communication, July 22, 2003.

8. T.H. Cormen, C.E. Leiserson and R.L. Rivest, Introduction to Algorithms. New York: The MIT Press, 1989.

9. W. Grosso, *Java RMI*. Cambridge: O'Reilly Media, 2001.

10. E.W. Weisstein, Frobenius Norm. *MathWorld—A Wolfram Web Resource*. http://mathworld.wolfram.com/FrobeniusNorm.html, last accessed October 11, 2006.

11. S. Gulde, "Experimental Realization of Quantum Gates and the Deutsch-Jozsa Algorithm with Trapped $^{40}Ca^{+}$ Ions." Ph. D. Dissertation, Institut für Experimentalphysik, Innsbruck University, Austria, March 2003.

12. K.-A. Brickman, P. C. Haljan, P. J. Lee, M. Acton, L. Deslauriers, and C. Monroe, "Implementation of Grover's Quantum Search Algorithm in a Scalable System." Phys. Rev. A 72, 050306, 2005.

13. J. Chiaverini, J. Britton, D. Leibfried, E. Knill, M. D. Barrett, R. B. Blakestad, W. M. Itano, J. D. Jost, C. Langer, R. Ozeri, T. Schaetz, and D.J. Wineland, “Implementation of the semiclassical quantum Fourier transform in a scalable system.” Science v308, p997-1000, 2005.

14. D. L. Moehring, M. Acton, B. B. Blinov, K.-A. Brickman, L. Deslauriers, P. C. Haljan, W. K. Hensinger, D. Hucul, R. N. Kohn, Jr., P. J. Lee, M. J. Madsen, P. Maunz, S. Olmschenk, D. Stick, M. Yeo, and C. Monroe, “Ion Trap Networking: Cold, Fast, and Small.” Proceedings of the XVII International Conference on Laser Spectroscopy, edited by E. A. Hinds, A. Ferguson and E. Riis, (World Scientific, Singapore) p421-428, 2005.

15. K.D. Underwood, K.S. Hemmert, "Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance," FCCM, 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04), p219-228, 2004.

16. G. Negovetic, M. Perkowski, M. Lukac, and A. Buller,"Evolving quantum circuits and an FPGA-based Quantum Computing Emulator." *Proc. Intern. Workshop on Boolean Problems*, p15-22, 2002.

17. M. Fujishima, "FPGA-based high-speed emulator of quantum computing," Field-Programmable Technology, December, p21-26, 2003.

18. A.U. Khalid, Z. Zilic, K. Radecka, "FPGA emulation of quantum circuits," ICCD, p310-315, 2004.

19. Nallatech FPGA Computing Modules. http://www.nallatech.com/?node_id=1.2.1&id=3, last accessed October 15, 2006.

20. Nallatech BenNUEY PCI 4E. http://www.nallatech.com/?node_id=1.2.2&id=2, last accessed October 15, 2006.

21. Nallatech DIMETalk FPGA Computing Design Tool. http://www.nallatech.com/?node_id=1.2.2&id=19, last accessed Oct 15, 2006.

22. Pragmatic C Software, Cver. http://www.pragmatic-c.com/gpl-cver/, last accessed October 15, 2006.

23. Aldec Active-HDL. http://www.aldec.com/products/active-hdl/, last accessed October 15, 2006.

24. L.M.R. Mullin, A Mathematics of Arrays, Ph.D. Thesis, Syracuse University, 1988.

Appendix A – FPGA Floating Point Formats

A consequence of storing data in SRAM is that it must be stored as an IEEE754 data type, shown in Figure 50. IEEE754 is a standard 32 bit floating point data type. This data type is also used for all data transfers between the FPGA and any external devices such as a host PC. All floating point computations on the FPGA are performed using Nallatech's floating point cores, however, which use Nallatech's own 38-bit float type, also shown in Figure 50. The slightly wider floating point type was designed to handle overflows that may occur during mathematical operations such as multiplications.

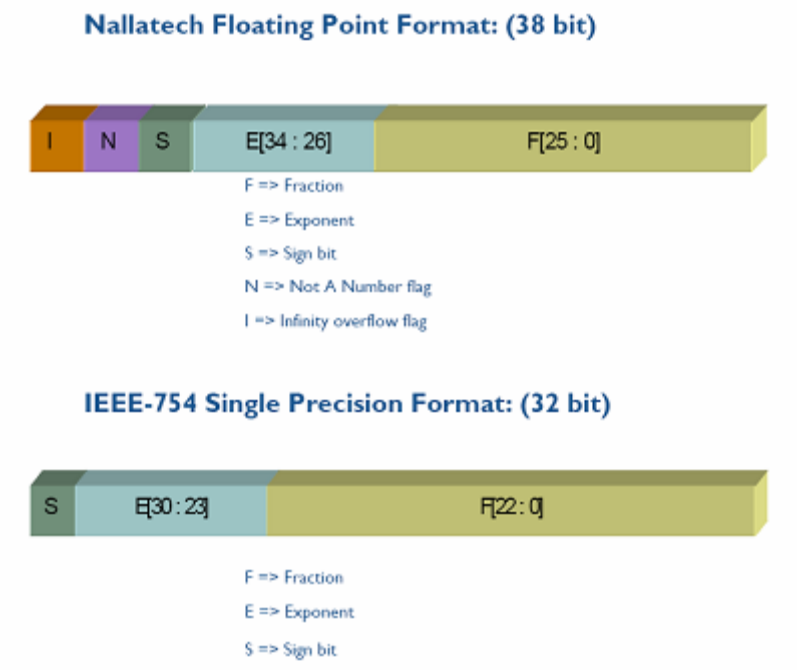

Figure 50: Floating Point Formats